Linköping University | Department of Computer Science
Master thesis, 30 ECTS | Information Technology
2018 | LIU-IDA/LITH-EX-A--18/036--SE

The evolution and erosion of a service-oriented architecture in enterprise software

– A study of a service-oriented architecture and its transition to a microservice architecture

---

En Service-orienterad arkitekturs erosion och evolution

Eric Karlsson

Supervisor : Jonas Wallgren
Examiner : Ola Leifler

LINKÖPING UNIVERSITY

Linköpings universitet
SE–581 83 Linköping
+46 13 28 10 00 , www.liu.se

Abstract

In this thesis project, a company's continuously evolved service-oriented software architecture was studied for signs of architectural erosion. The architecture has been continuously developed over some time and the company have experienced a reduction in architectural quality and felt that it no longer fulfilled its design goals and therefore decided to start working on a replacement architecture based on the microservice architectural style. This thesis project therefore aimed to study how the current architectures quality have changed during its evolution, find the causes of these changes in quality, and estimate how the planned microservice migration will effect these changes in quality.

This study was performed in three steps. First, a suite of suitable quality metrics where gathered based on the stated architectural design goals and what information can be extracted from the history of the implemented architecture. A tool was developed to model the architecture and to gather the quality metrics from the current architecture and how it has changed over one year's worth of development and evolution. Secondly, the causes of these changes in architectural quality was investigated through developer interviews with a wide range of developers that had worked on the architecture and the web application that it provides the structure for. The interviews focused on the topics of architectural knowledge, what consideration is taken to its design during component development, maintenance of existing components and architecture, as well as questions regarding specific components and anomalies. Thirdly and finally, the migration to a microservice and its effects on the quality of the current architecture is estimated through performing microservice reengineering on the model used to evaluate the current architecture.

The tools developed during this thesis allowed for an analysis of the architecture did show an increase in consistency violations, structural problems and level the of coupling have substantially increased over the version history that the model tracked. It was discovered by the developer interviews that some of the causes of this erosion was due to among other reasons an abandonment of some architectural deign decisions, lack of architectural knowledge on certain topics, and none-optimal development conditions and priorities. The microservice reengineering showed how the migration could be used to improve the measured quality metrics and that a migration alongside some other architectural erosion prevention and repair methods could create an architecture that are more modular and erosion tolerant.

Acknowledgments

I would like to thank Niklas, Annika and Marcus for the opportunity to conduct this thesis project and the guidance needed to complete it as well as everyone who volunteered their time to participate in the developer interviews.
I would also like to extend the same thanks to Jonas and Ola for all the invaluable feedback and support, which without this thesis project would not have been possible. Additionally, I extend a big thanks to Per, Göran, Anders and Torsten for answering all my questions, both the dumb ones and the difficult ones. Lastly, I would like to extend a special thanks to Filip, Viktor, Johan, Ted, Jasmin and Viggo for all the support, encouragement and levity during this thesis project. It would not have been as fun without you.

Eric Karlsson
Summer 2017

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

The microservice architectural style has been gaining popularity over recent years alongside the rise of the DevOps and continuous integration rapid development style [6]. The microservice architectural style promises a flexible, independent, modular and easily expandable set of software components called microservices that together can power any number of applications. These properties have convinced many high-profile companies, like Netflix and the BBC to switch to using microservices to power their backend architectures. For some, the choice of microservices architecture is made as a replacement architecture to the similar architectural style service-oriented architecture [16]. This transition is often undertaken with the belief that a microservice architecture will increase the autonomy, modularity, testability as well as lower the coupling of the architecture's components. That in turn will create a more erosion tolerant architecture, that will allow the developers to add, change or replace services as functionality with minimal impact on other architectural components. But the question remains, what steps have to be made to ensure this high erosion tolerance? What lessons can be learned from the implementation of the similar service-oriented architecture? How can a migration from service-oriented architecture to microservice architecture be used to address issues that have through architectural erosion entered the service-oriented architecture? What are the actual differences when assessing these architectural issues when using the microservice architectural style? The contents of this report are based on a study that have been performed on a Swedish company's service-oriented architecture that hosts an enterprise web application. The developers at the company are in the process of transitioning from a service-oriented architectural solution to a microservice architectural solution for their enterprise software.

1.2 Aim

This thesis project intends to study and measure the architectural erosion and evolution of an enterprise implementation of service-oriented architecture. The architecture and the applications using it have been continuously developed for several years. This report outlines what effects this continuous development has had on the overall architectural quality as well as to observe its erosion and what steps have been taken to mitigate this during its continuous development. This thesis project also intends to examine how architectural compliance, structural health and coupling can be assessed through a method based on the reflexion model. This thesis finally intends to perform and measure the effects of a reengineering process of a service-oriented architecture migrated towards a microservice architecture.

1.3 Research Questions

In order to fulfill the aims that was stated above the following research questions were set up as a basis for this thesis work.

1. ***How can architectural evolution and erosion of architectural qualities be measured when analyzing a continuously developed service-oriented architecture?***
2. ***What architectural problems and challenges appears when continuously developing a service-oriented architecture based on OSGi?***
3. ***What effects can a migration towards a microservice architecture have on this issues and overall architectural quality?***

1.4 Delimitations

In order to keep the scope of this thesis at a manageable level the following delimitation have been outlined.

1. ***One architecture implemented at one company was studied.***
2. ***The changes of the implemented architecture was only tracked over one year of history.***

2 Theory

2.1 Software Architecture

While software architecture and its design have become an essential part of almost every large software project a universal definition has not been agreed on within the industry [15]. In fact, a list [20] has been created and is maintained by the Software Engineering Institute [21] to gather definitions of software architecture from the software development community. While a universally accepted definition of software architectures does not exist, most definitions overlap in their focus on the description of the relationships that exists between software components that exists within a program or software system. Where an architectural component is a grouping of functionality, software packages, web services or any semantically related code that can is cohesive enough to be abstracted into a module. With this in mind, the definition that seems most useful for the purposes of this thesis is the one provided by the IEEE organization.

"Architecture is defined by the recommended practice as the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution." [18]

It is obvious from the definition above that software architecture is an abstraction of implemented or planned software. An architectural component does not directly map to a source code file but is rather a partition of functionality that the software system is expected to provide. The component is the primary unit of abstractions used when describing architecture. A component is designed by an architect to be responsible for a certain functionality within the architecture, for instance an architectural component may be responsible for providing the means to read from and write to a database. The architecture also define what relationships exists between components, this defines the structure of the architecture and it is important for the quality of the architecture that these relationships, or dependencies, are well constructed in the architecture and followed in the implemented software. To ensure that components are appropriately separated and interact in well-defined ways. In the aforementioned database component example, the architect would have to outline what other components should be able to make use of the functionality that the database component provides, as well as how those components accesses that functionality. This relationship property is important to support the last section of the IEEE definition, that an architecture should provide the principles of its design and evolution. Having loosely coupled architectural components, i.e. having few and well-defined dependencies between components, eases the process of further change the architecture in to address new requirements later in its life time as well as easing in the process of implementing the architecture as actual software [15, 18]. It is also important to point out the word "environment" in the IEEE definition, the software architecture also outlines what requirements are placed on the environment that the software system is designed to run within. Environmental requirements can be as simple as defining what operating systems the architecture will support but they can also have more complicated requirements such as how certain components are expected to run on different hardware. The client-server architectural style is an example of an architecture where components are expected to be able to run on different machines and interact through a network.

2.1.1 A few architectural terms

This section will define a few software architectural terms that will be used many times, throughout the thesis report. The *designed architecture*, also known as the *intended architecture*, is the theoretical design that have been developed by one or more architects often before any development has started. The *implemented architecture* is the actual implementation of the designed architecture and may differ from the theoretical design. This report also touches the topics of creating models of architectures. These models are abstract representations of either a designed or implemented architecture often by representing the architecture as a graph where architectural components are shown as nodes and the relationships such as dependencies or inheritance are shown as edges between the nodes. The methods used for this modelling process is outlined in the method chapter.

2.2 Software Architectural Quality Attributes

The choice of architecture has to be made with care to ensure that the chosen architecture serve to provide the desired architectural quality attributes [15]. Architectural attributes serve to ensure that the application fulfill its non-functional requirement.

2.2.1 Scalability

Scalability is a measurement of an application's ability to scale along with the size of the demand that is placed on it [15]. This change in scale should not in any way affect the user experience or negatively affect any other quality attributes [32].
The nature of this attribute changes depending on what kind of application that is being developed but in the context of web applications it usually refers to how the application responds to increases in simultaneous connections and the number of requests on the application as well as increases in the size of the data it is expected to process [15, 32].

2.2.2 Modifiability and Modularity

An architecture's modifiability is a measurement of how expensive, in developer time and effort, changes to the application is to make during its lifetime [15]. In some architectures, it is hard to give an exact measurement of how susceptible it is to be modified, in most cases modifiability has to be estimated [15].
In some architectural styles, namely service-oriented architecture [32] and microservices architecture [31], components provide their functionality through so called services are used

to separate the architectural components into more or less independent architectural components. This is done for multiple reasons, but it is primarily to allow developers and architects to more easily modify and add components with as little impact on the overall architecture as possible.

2.2.3 Availability

Availability is a measurement of what ratio of time that the application is available out of the time it's expected to be available [15]. Architectural approaches to deal with the challenge of availability can be aspects such as *fault tolerance*, the ability for an application to handle errors. *Recoverability*, how well a crashed application can recover from failure. And *replicability*, the ability of multiple instances of the same components to be available in cases where one of them are not available.

2.3 Software Architectural Erosion

The architectures mainly examined in this report are designed to allow for continuous changes during the applications lifetime. These changes are made to allow the applications to fulfill new functional requirements. To make use of new technologies or to fix issues that had not been detected during testing. However, continuous changes to an application or the underlying architecture without the careful consideration of the architectural structure and design can lead to the diminishing of architectural quality. This process of cumulative continuous changes to an architecture leading to diminishing architectural quality is called *software architectural erosion* [9]. It is important to differentiate the term architectural erosion from the term architectural drift [9, 30]. Architectural drift describes the differences between implemented architecture and designed architecture that is caused by updating the implemented architectures. Drift as a concept is not concerned with the qualitative impacts of these discrepancies while architectural erosion concerns the impact of these discrepancies on the qualitative goals of the architectural design.

The cause of architectural erosion is usually modification or evolution of the architecture without proper consideration to the architectural and design rules outlined in the designed architecture. Common areas of erosion are:

- **Architectural integrity**
  The integrity [9] of a implemented architecture is a measurement of how well it conforms to the designed architectures requirements on what components need to be implemented as well as what relationships they are expected to have.

- **Quality attribute conformance**
  During the design phase of architectural development an architecture is designed to fulfill some qualitative attributes such as a certain level of scalability, modularity or coupling between components. If these design decisions are not considered during the maintenance or evolution, see section 2.4, of the architecture they risk being eroded over time.

2.3.1 Causes of Architectural Erosion

The causes of Architectural Erosion are varied, but are all related to introducing new or updating code or software components into to an existing architecture in an improper way as related to the architectures design [11, 48, 50]. The causes can be further specified into the following categories:

- **Unmodifiable Architecture**
  The planned changes may not be supported by the architecture they are planned to be introduced into without performing major reengineering of the original architecture. Also, if such reengineering is not done properly new architectural design goals can be introduced into the architecture that contradicts with existing architectural design that can further erode the architecture.

- **Violations of architectural design principles during development**
  If a planned change does not take the architectures design into account, then those changes can erode the overall architectural integrity. Common reasons for this is an *uncertainty about intended evolution* where developers are uncertain of or disagree on the intended evolution of the architecture. *Radical changes of requirements* where major or frequent changes of the architectures requirements can lead to a disjointed design.

- **Improper implementation of changes**
  When changes to the architecture are implemented under non-optimal development conditions they will risk introducing erosion into the architecture. Such as non-optimal development conditions include *poorly defined requirements*, where functional and non-functional requirements do not take the current architecture into account. *Inadequate development environment*, if development tools, architectural documentation or version control systems are inadequate then developers implementing planned changes can introduce architectural violations. *Inadequate developer resources*, when not enough developers, skilled developers or development time are assigned to implemented planned changes the number of architectural violations can increase. *Release pressures* are also a contributing factor, either through tight deadlines, poorly timed release cycle or unsynchronized releases between development teams.

- **Lack of Architectural knowledge**
  The lack of architectural design knowledge will prevent developers to make development decisions that are in line with the intended architectural design. Causes of this lack of knowledge can simply be a lack of proper or up to date *Architectural Education*, or *Architectural Documentation*.

- **Accumulation of complexity**
  As architectures are iterated upon and evolved they often increase in complexity. This can lead to architectural maintenance tasks become more expensive in time and effort. This coupled with an iterative development methodology that prioritizes fulfillment of new functional requirements over maintenance or re-factoring of older code can lead to further erosion of architectural quality.

2.3.2 Effects of Architectural Erosion

The effects of software architectural erosion are most often not the failure to fulfill functional requirements or a decrease in performance. The effects are rather on the ease of evolving and maintaining the architecture as well as its fault proneness [9]. Some of the effects of software architecture is outlined below [9, 11, 48, 50].

- **Increased maintenance cost**
  As mentioned above, one of the common effects of architectural erosion is the increased cost in time and effort of maintaining the architecture when multiple components are in violation of architectural design directives.

- **Increased design confusion and uncertainty**
  When the architectural design has been muddled due to contentious erosion the existing code base will no longer serve as a good example set for how to develop for the

architecture. As newer developers learn to work within the architecture they often take the wrong lessons from the existing code as to the intended architectural design. The impact of this problem will be further increased if many design violations exists within the architecture and no up to date specifications or documentation are available to the developers.

- **Erosion of quality attributes**
  When changes are introduced into a software architecture that are not consistent with the architectures stated design goals or the quality attributes that the goals where design to fulfill. The quality of the code will often suffer, and the rate of defects and regression problems will often increase.

- **Increase in component complexity**
  A non-uniform architecture will lead to increased complexity in components. These complexities come from the increased number of interfaces and dependencies required for components to fulfill an ever-expanding set of functional requirements.

2.3.3 Mitigation of Architectural Erosion

The approaches to mitigate erosion can be divided into erosion *minimization*, *prevention* and *repair* [9]. Minimization methods are focused on ensuring that development and evolution of the architecture comply with its stated design. Erosion minimalization include methods and processes for keeping architectural documentation and design up to date, providing architectural knowledge to developers and architectural conformance and dependency monitoring as well as code and pattern generation. Prevention methods are used to remove cases of erosion and includes methods for monitoring and enforcing architecture compliance at runtime and automatic architectural self-adaptation. Erosion repair methods, sometimes known as erosion reversion methods, are methods that focuses on restoring already implemented code to architecture compliance and includes methods for extracting an architecture from implemented code, discovering the intended architecture when documentation is unavailable, and realign an implemented architecture with its intended design.

2.4 Software Architecture Evolution

The task of maintaining, changing and developing already existing software architectures in order to fulfill new requirements or better meet already stated requirements is called software architectural evolution [7]. There exist several different methods that facilitate architectural evolution, inducing but not limited to *code refactoring*, the task of restructuring code to better conform with the architectural design without changing its external behavior. *Maintenance*, the task of making sure that code and components adhere to architectural design and are free from any faults as they are changed. Finally, *architectural reengineering*, that is outlined in the section below.

2.4.1 Reengineering

Architectural reengineering is a method for redesigning existing implemented architectures. This is done through a process of extracting a model of the implemented architecture, using the model to reason about planned changes, and finally implemented these changes creating a new implemented architecture. [23]

The Horseshoe Model

A common approach for reengineering existing architectures is the horseshoe model [7]. While this model is often extended with additional steps to match the level of reengineering planned or the desired target architecture [17] the steps of this method is outlined in figure 2.1 and is listed below.

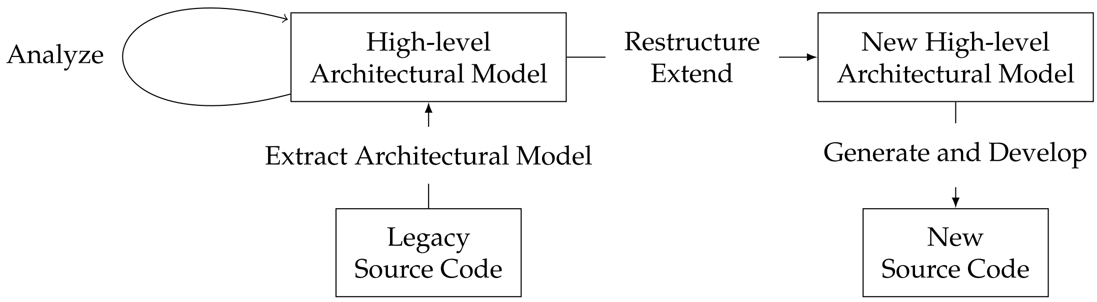

Figure 2.1: The Horseshoe Model adapted from [7, 17]

1. **Extract an architectural model**
   A model of the legacy architecture is constructed. The model is an abstract representation of the written code and the representation may differ depending on the needs of the architectural design. However, the most common approach is representing the code as a diagram of nodes representing code components and edges between representing dependencies, inheritance and other relationships. This model can be manually constructed by architecture experts, extracted from the source code using modeling software or derived from architectural design documentation. If architectural documentation is used as a basis for this model then it is important to first confirm that the implemented architecture conforms to the designed architecture.

2. **Analyze**
   A architectural analysis is performed in order to determine what architectural changes are needed for the architecture to fulfill new requirements or architectural styles.

3. **Restructure the architectural model**
   Existing architectural components are restructured or removed, and new ones are added to conform to new architectural requirements.

4. **Develop new source code**
   Legacy code is adapted and new code is developed to implement the new architecture. If possible, code generation tools can be used to create software components conforming to the new architectural model.

2.5 Architecture Analysis and Evaluation

Depending on what literature you consult the definition of Architectural analysis can vary greatly and the term is often used interchangeably with the term architectural evaluation [54]. In Dobrica and Niemeläs well cited survey [10] architecture analysis is described as a method for evaluating the quality of a proposed architecture in order to predict what effect it will have if implemented. This is usually done through either a *questioning technique* or a *measuring technique*. When analyzing an architecture using a questioning technique a set of questions are developed, usually in the form of scenarios or checklists, that are applied to a proposed architecture to see if it has the desired properties. Measuring techniques on the other hand strive to develop measurements from a proposed architecture, generally though simulations or prototypes, that intend to measure specific software attributes. While these techniques are useful for detecting potential erosion risks when evaluating proposed architectures, they are not suitable for evaluating the current state of an implemented architecture. In this thesis report architectural analysis will refer to the act of evaluating how well an implementation conforms to the designed architecture as well as the designed quality attributes as outlined in the architectures documentation. This type of evaluation is known as *Static evaluation* [26]. Static evaluation is used for several different purposes [26], form detecting the reuse potential of components in one architecture in another to extracting the basis for architectural documentation. In this thesis, static evaluation will be used to assess the *consistency*, a measurement of how well the implemented architecture matches the rules outlined in the designed architecture, and to detect potential structural problems.

2.5.1 The Software Reflexion model

The reflexion model is a method for assessing the level of erosion or architectural drift, present in a implemented software architecture [30]. The purpose of the method exploiting the differences in implemented and designed architectures for checking architectural conformance, update documentation or detect architectural problems. The method is based on generating two architectural models. One model is based on the *designed* architecture, sometimes called *high-level model*. This high-level model is generated from documentation or manually created by software architects and represents the architects intended architecture. The other model is based on the *implemented* architecture which is statically extracted from source code or dynamically extracted through monitoring the execution of the software that is being modeled. These models consist of modules, that represents the software components, and calls that represents how the components interact. The modules and calls are often illustrated as vertices and edges in a graph.
The final model generated is the *reflexion model*, this model is created by comparing the high-level model with the implementation model. All the modules and the relationships between them are mapped unto the reflexion model but is labeled according to the following rules:

1. **Convergence**
   A relationship present in the high-level model that is also present in the source model is considered to be *convergent*.
2. **Divergence**
   A relationship that is present in the source model but is not in the high-level model it is considered to be a *divergence*.
3. **Absence**
   A relationship that is present in the high-level model but is not present in the source model is considered to be *absent*.

While fairly simplistic, the software reflexion model can serve as a basis for many different software development tasks including, reengineering, architectural conformance checking and updating of outdated documentation [30].

2.5.2 Static Evaluation

There exist many static evaluation methods. The before mentioned *Software Reflexion model* method [30] that uses high level representation of the designed architecture and compares that with a source model extracted from the implemented architecture to create a reflexion model that outline where they do not agree. The *Hierarchical Reflexion Model* method [27] that extends the reflexion model to include a hierarchical view of components. There are methods that focuses on *architectural conformance checking* through the examination of dependencies using *dependency-structure matrices* that analyze the relationships between components through

what dependencies are allowed and not according to the designed architecture [36]. The *source code query language* method allow developers to express and check architectural constraints as queries [36].
While these methods have not been expressly developed with service-oriented architectures in mind they can still be applied as the SOA architectural style does not introduce any architectural feature that prevents these methods to be applied. However, one of the few evaluation methods that focuses on the SOA architectural style is the LISA architectural design language, created by Weinreich and Buchgeher. The LISA language presented is presented in the paper *Towards supporting the software architecture life cycle* [54]. Weinreich and Buchgeher present the tools, methods, activities and metrics for managing the design, implementation, maintenance, evolution and static analysis of implemented architectures. The steps for architectural analysis that LISA method make use of are listed below:

- **Consistency analysis**
  If a relationship between components are valid according to the rules outlined in the architectural documentation it is said to be consistent. Therefor consistency is a measurement of how many of the relationships, mainly dependencies, that are present in the implementation are valid according to the architectural documentation.

- **Completeness analysis**
  Completeness is a measurement of how well the components present in the implemented architecture fulfill the requirements placed on them in the architectural documentation.

- **Conformance analysis**
  Architectural conformance is a measurement of how well an implemented architecture follows the designed architecture in terms of what components are present or absent as well as what valid or invalid relationships exists between them. Conformance is measured by comparing implementation to the overall architectural documentation to detect discrepancies between them. The discrepancies measure are the presences, absences and divergence of components and the relationships between them. This is done to mitigate architectural erosion and to make sure that architectural documentation is up to date.

- **Structural analysis**
  Structural analysis aims reveals structural problems in the implemented architecture. Proper architectural structure is often based on best practices and the experience of the developers and architects. The structural health of an implementation is measured through performing checks that have been developed with the architecture in mind. Such checks can measure dependencies between modules to reveal structural weaknesses, such as cyclic dependencies, architectural layer violations and checking for whether a service is accessible, also called visible, from outside the architecture that should not be. These checks are some of the more common structural analysis checks, but this analysis step in the LISA method is designed to be customized for the needs of design goals of the implemented architecture.

- **Decision analysis**
  By mapping architectural design decisions to the impact they had on the quality on the architecture an architect can validate if they had the desired effect. This process is largely done manually by architects.

Weinreich and Buchgehers have created these steps for architectural analysis to complement there LISA architectural design language [54], and while the tools developed for the LISA model nor the model itself will be used in this thesis work the steps outlined above provide a good set of metrics and approaches for performing static evaluation on service oriented architectures.

2.6 Monolithic Architecture

The architectural styles of service oriented architectures and microservice architecture is often described in ways how they differ from monolithic system architectures [31, 41]. A monolithic architecture, as shown in figure 2.2 below, is characterized as an architecture where all the components and functionality are deployed as a single piece of software or a few large connected software components.

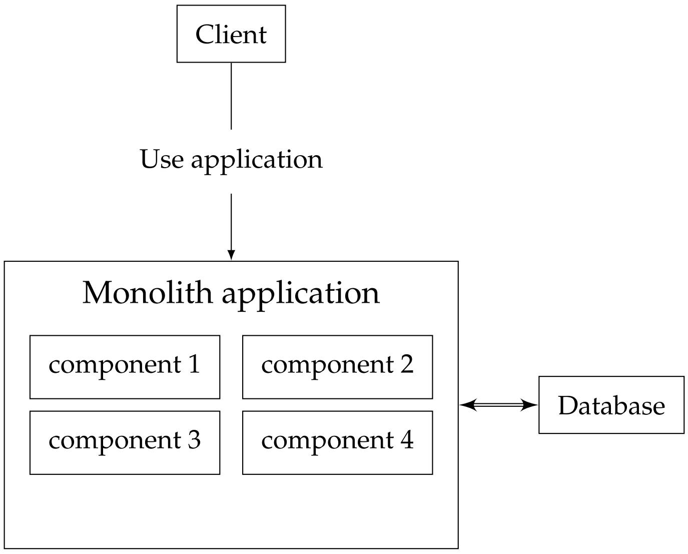

Figure 2.2: Monolithic Architecture adapted from [34]

While the components may be loosely coupled they are not independently deployable and are therefore not able to function without the rest of the components in the architecture. Changes to one will require the whole architecture has to be redeployed in order for those changes to take effect. When a service-oriented architecture deteriorates it is said to gain monolithic attributes, this usually refers to the inability for the components to be deployed and updated interdependently of each other.

2.7 Service-oriented Architecture

Service-oriented architecture (SOA) is the architectural implementation of the Service-oriented computing (SOC) paradigm [33]. This architectural style divides its software components into services. A service is an independent piece of software that provide some specific functionality, and should perform its function without needing to be aware of other services states and the larger application context [34]. While services should have independent executions, they are still expected to be able to communicate execution requests, also known as invocations, with each other using publicly available interfaces. By allowing services to communicate they can accomplish more complicated tasks requiring the functionality from multiple services. Services provide the functionality, but they are not by themselves an application. Applications in SOA instead use the architecture to consume the functionality of one or more services, often through some sort of front-end module [33]. While each architectural implementation of SOA differs depending on its intended application, they generally have been designed with the primary characteristics of modular independent components. The components are typically dynamically consumable and discoverable by other components. The implementation and hosting location of the components are independent of each-other. Leading to architecture made up of loosely coupled components [47].

2.7.1 A Service in SOA

In SOA services are the modules that provide the functionality used by architecture's applications or by other components within the same architecture. A service is a self-contained module providing some functionality through a well-defined interface [34]. Depending on the implementation of SOA the architecture's components can provide multiple services per architectural component, as well as allowing components to make use of other services to create new composite services. A component that make use of a service is referred as its consumer and can be another SOA component within the same architecture or outside components that can access its interfaces. A service implemented in SOA has the following characteristics:

- **Implementation Autonomy**
  A service should be invokable though a well-defined interface, this interface should outline what the service consumer is expected to provide and what the consumer can expect to get returned to it. The services internal implantation should however not be viewable from outside the service. This characteristic gives the service developers the freedom to develop the service using any programming language or using whatever technology they feel are the most appropriate for the task the service is expected to perform [34].

- **Loosely coupled**
  The usage of a service should not require any knowledge of its implementation, internal structure or current state. This ensures that services are loosely coupled even if they are used in the same application or have dependencies on each other [33].

- **Location independence**
  A service should function wherever its run in the same environment as the application or on some other environment reachable through the network or internet [33]. The implemented architecture should provide the means to find and discover services [34].

The objective of these properties is to provide a modular, re-usable and easily maintainable piece of software. Multiple applications can use the same services, a service can be updated and iterated upon or removed without effecting another service [34].

2.7.2 The Architecture of SOA

To allow applications to take advantage of services they need to rely on the architectural implantation. A SOA architecture is expected to provide the following features:

- **Service discoverability and registration**
  The architecture must take into account how the service consumers find the services they need to consume. The architecture also need to provide methods for services to register themselves as available or no longer available to consumers [34, 45].

- **Intercomponent communication**
  What protocols and technologies are used for communication between services and service consumers [45].

- **Service access or invocation interfaces**
  The architecture has to provide common interfaces to allow services to be invoked, messaged or to handle events [45].

- **State transfer**
  To create an architecture with the loosest possible coupling between components statelessness should be enforced as much as possible. However, when a state need to be preserved between services the architecture has to provide the protocols and infrastructure to transfer and track them [45].

- **Service composition**
  To provide certain functionality services may have to be combined into a new composite service [34]. The architecture has to provide ways for developers to do this, as well as methods for changing service interservice behavior when used in a composition [45].

Service Brokering

Service brokering is an architectural solution to the problem of allowing services finding each other and to know when services are available. This is done with the following three components [34], which interactions are illustrated in figure 2.3.

- **Service provider**
  This component provides one or more services as well as rules for what kind of consumer are allowed to access them to a service broker.

- **Service Registry**
  This component handles incoming requests for services and redirect them to the appropriate available service provider. This component also retrieve notifications from service providers about what services they provide as well as when they are available. This component is also sometimes called Service broker [33].

- **Service client**
  This component consumes one or more services and put them to use. This component is sometimes called Service consumer [33].

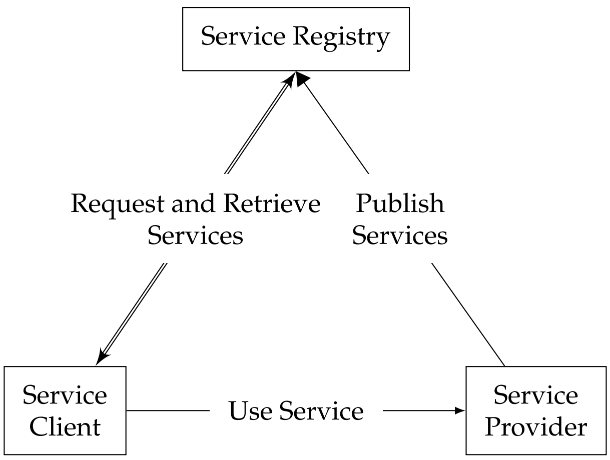

Figure 2.3: Service Discovery in SOA adapted from [33, 34]

2.7.3 Metrics for Evaluating SOA Design

Since the structure of a service-oriented architecture is not directly comparable to object-oriented or other more traditional architectures and therefore, the metrics and methods that have been developed to measure them do not directly apply to service-oriented designs [38, 53]. This is because the object-oriented concepts of classes and methods are not directly applicable to services [38], that are themselves another layer of abstraction above methods and classes. The lack of inheritance in SOA and the more granular nature of code sharing through service consumption, where one architectural component can provide multiple services and consume multiple services from multiple sources creating new services. This does not mean that metrics traditionally used for evaluating object oriented designs are not useful for detecting and predicting potential problems and anomalies within a SOA design [53], they simply have to be adjusted to better measure the characteristics that are desired in a SOA design. When evaluating SOA design the focus is usually put on *Coupling* and *Granularity*.

Coupling

The level of coupling in a service oriented system is a measurement of how tightly connected an architectures components are. In SOA a loosely coupled design is desirable since it makes the architecture more modular and modifiable, a service can be changed or removed with less effort if it effects fewer other components. Coupling is measured by analyzing the dependencies that exists between services [43, 53], a service with few and well-defined dependencies is considered loosely coupled.

Granularity

Granularity is a measurement of the size and complexity of the services within the design through the number of operations they have [53]. This essentially means that if a service provides a large number of functions it will have more service consumers, meaning that changes to that service will affect more components. Services with a coarse granularity will make the architecture harder to maintain, but if the granularity of services are too fine then the functionality will have to be spread out over multiple services and that will contribute to higher coupling.

2.7.4 OSGi

Open Services Gateway initiative framework (OSGi) is a framework based on the specifications developed and maintained by the OSGi alliance [4], and is a way of implementing the service-oriented architectural style in practice. The purpose of the frameworks is to provide for, manage and support components, called bundles, for implementing services developed in Java [46].

OSGi Bundle

Bundles are the modular components that provide the services in the OSGi framework. Bundles are essentially Jar files with regular Java classes, data and resources that are necessary in order to provide the bundles functionality. This regular Java functionality is bundled together with additional OSGi specific metadata. The metadata is stored in a manifest file [3] that is included with every bundle and outlines among other things what Java packages the bundle provide and imports as well as what services the bundle provides and is dependent on.

OSGi Service Registry

In order to manage the life cycle of OSGi bundles within the architecture functionality for dynamically adding, removing and replacing OSGi bundles at runtime has to exist. As well as functionality for bundles to dynamically to find and consume services provided by other bundles. In OSGi this functionality is provided by the a service registry [2]. This service registry is the implementation of the service brokering requirements of SOA architecture as outlined in section 2.7.2. The service registry makes sure that services can be found by the bundles that need them, this is done either through having service consumers search the service registry for the services they need or through having service consumers subscribe to notifications from the service registry that outline what services become available or unavailable. The interactions of these components are shown in figure 2.4 below.

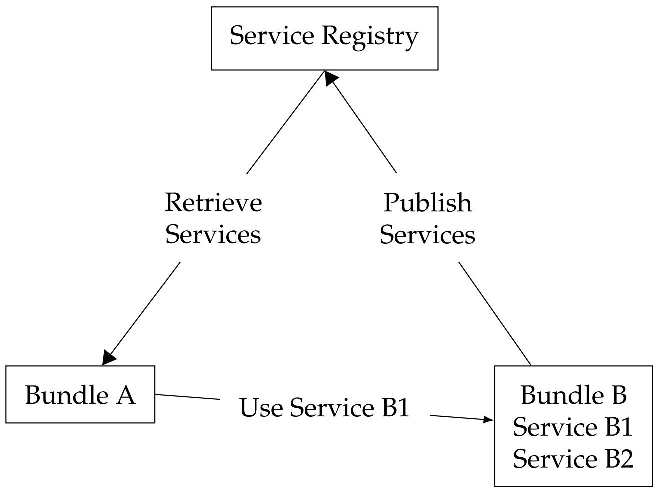

Figure 2.4: Service Registry in OSGi adapted from [46]

2.8 Microservice Architecture

The microservice architectural style, illustrated in figure 2.5, has been described as "old wine in a new bottle" [44] or "just a more specific pattern of SOA" [37, 41]. A universally accepted definition of microservice components and the architecture that facilitates their use does not exist. However, most literature have described microservices in the following ways.

- **Autonomous services**
  Similar to services in SOA, microservices are self-contained module providing some functionality. However, in microservice architecture a microservice should have as few as possible dependencies on other components and support systems in order to ensure the autonomy of the microservice component. A common consequence of this is that microservices often provide their own support systems like databases and interfaces to third party systems instead of having a dependency on another component for this functionality.

- **Loose coupling, High cohesion**
  As mentioned above, a microservice component should be as loosely coupled to the rest of the architecture as possible. However, the cohesion within a microservice should be high. Meaning that a microservice should contain as much related functionality as possible meaning that if some architectural feature needs to be changed ideally only one microservice needs to be updated [31].

- **Accessible through lightweight interfaces**
  A microservice should be usable through a lightweight interface. These interfaces are usually in the form of a stateless API that are implemented using the Hypertext Transfer Protocol (HTTP) [14], Representational state transfer (REST) [13] or Simple Object Access Protocol (SOAP) [29] protocols. These protocols allow for simple and when possible stateless request-response type communications to allow service consumers to invoke service functions and receive needed output while limiting the consumer to service coupling.

- **Automatically deployable and movable services**
  The microservice architecture further stresses the independence and loose coupling of services in SOA by requiring them to be fully independently deployable [41], movable and duplicated by the environment at any time. This means that a microservice cannot have any dependencies on local resources.

- **Duplication of services**
  In order to allow for automatic scaling of microservices the functionality it provides should not be affected by having multiple instances of the service running at the same time. This can present challenges to services that handles session data or require persistence [41].

- **A fine grained implementation of SOA**
  While most properties of a microservice architecture are directly inherited form the similar to SOA [16]. The differences are primarily in the increased autonomy of services. Dependencies between services are also generally more discouraged and if necessarily should be done through REST or HTTP APIs.

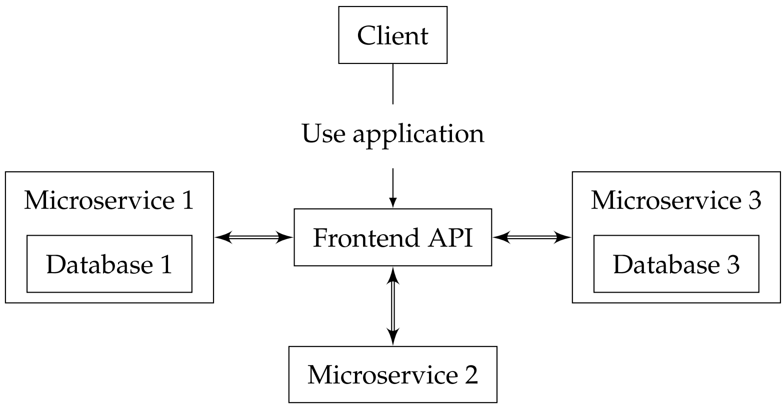

Figure 2.5: Microservice Architecture adapted from [41]

Service discovery and consumption is handled in similar ways as in SOA implementations. It is important to note that the "micro" in microservices generally does not refer to the size of the code it's made up of or the size of the functionality the service provides [16]. The Micro instead refers to the impact the service has on the architecture through its high autonomy, meaning that that the impact of changes or replacements of a service should be small on the architecture or on other services. This independence property can therefor serve to create an even more loosely coupled, modular and flexible architecture compared to the similar SOA architectural style [16].

2.9 Case Study Methodology

Case study methodology for software engineering as described by Runeson and Höst [40] is an adaptation of a common research methodology used by researches within the areas of political sciences, psychology, business research and other areas of research. The purpose of a case study is to empirically investigate a contemporary phenomenon in its context. The case studies are usually defined as one of four following types:

1. **Exploratory Studies**
   The purpose of exploratory case studies is to find new insights in some area of study in order to generate new hypotheses for future studies.

2. **Descriptive Studies**
   The purpose of descriptive studies is to describe and portray some phenomena.

3. **Exploratory Studies**
   The purpose of exploratory study is to find an explanation to a problem or phenomena.

4. **Improving studies**
   The purpose of an improvement study is to attempt to improve some aspect of a studied phenomenon.

Empirical research and case study methodology follow the following steps.

1. **Study Design**
   The objectives of the study are defined and the study is planned.

2. **Data collection preparation**
   The protocols and procedures used to collect data for the study are defined.

3. **Data collection**
   The data is gathered using the prepared procedures.

4. **Data analysis**
   The collected data is analyzed.

5. **Reporting**
   The results from the study is gathered and reported.

These steps do not have to be performed sequentially. For example, if the researchers determine that insufficient data have been collected when the data is analyzed the researchers are free to return to the data collection step to collect more information.

2.9.1 Triangulation

In order to assure the precision of empirical studies and case studies it is important to provide many different angels on the phenomena under study. If multiple angels point towards the same conclusions then the conclusion is considered to be more accurate. This concept is called *triangulation* and can generally be applied in the three different ways shown below:

- **Data source triangulation**
  The usage of many data sources and the collection of this data at different times will further strengthening the conclusions that are made from the data.

- **Multiple observer triangulation**
  Using multiple observers reduces the chances of one observers biases to influence the results.

- **Multiple methodology triangulation**
  The usage of multiple data collection methods and triangulating the data collect will further strengthening conclusions.

2.9.2 Data Collection Methodology

The data collection methodology is the core of the data collection step. Different types of data collection methodology are outlined and organized based on the level of direct contact the researches have on the data collection subjects below:

- **First degree methods**
  In first degree methods are data collection methods where the researchers are in direct contact with the subject under study and collect the data in real time. Examples of such methods are interviews and focus groups.

- **Second degree methods**
  In second degree methods are indirect data collection methods where the researchers collect data form the subject under study without interacting with them. Examples of such methods are recoding subjects using software or performing a task.

- **Third degree methods**
  Third degree methods are methods where researcher use already collected data or artefacts for study and therefore the researcher has no contact with the subject under study. Examples of such methods are gathering of metrics from documentation or databases.

Interviews

Interview based data collection is one of the more common first-degree data collection methods. As the name implies, interviews based methods are conducted by researches asking a subject or group of subject's questions relating to the case under study. The researches conducting the interviews have to design a set of questions that serve to provide the insight, context or data needed to answer the case studies research questions. The questions can be written to be *open* or *closed*. *Closed interview questions* are designed to allow the interview subject to select its answer from a limited set of possible answers, similar to answering a survey. *Open interview questions* allow the subject to answer the question however it feels are the most appropriate, with no restrictions. There are also a few different interview structures that the researchers can make use of, *unstructured interviews*, *semi-structured interviews* and *fully structured interviews*. Fully structured interviews only use pre-written questions and the order are asked are planned before the interview is conducted. Unstructured interviews use no pre-written questions and instead a general outline of areas of interest is used as a basis for the conversation between the interviewer and the subject. Finally, a semi-structured interview structure uses a set of planned questions, but the order of the questions is asked is not pre-determined. The semi-structured structure also allows for the interviewer to ask improvised follow-up questions based on the subject's answers.

3 Background

This chapters serves to outline information about the architecture that will be used as a test case for this thesis report. A full overview of the architecture, its components and the platform that hosts its components cannot be provided. Since such information risks the confidentiality of the company's software. This chapter instead strives to provide enough information about the architecture and its components so that you, the reader, can better understand the methods, results and conclusions presented in this thesis.

3.1 The Architectural Platform

The software platform hosting the service-oriented architecture implemented at the company uses a combination of proprietary in-house developed software, open source software and third-party software to provide the environment that hosts the architecture. While the hosting and container platform used has an impact on the implemented architecture the implementation of this platform is outside the scope of this report, however it is important to note a few attributes of the architectural platform:

- **Components as OSGi bundles**
  The software components responsible for hosting services within the architecture are made up by a couple of sub-components each consisting of one OSGi bundle. These bundles are constructed in accordance with the OSGi framework [2].

- **Logical Layering**
  Three architectural layers are defined within the architectural documentation. Bundles hosted by the platform are placed within one of these layers when designed. However, these architectural layers do not have any functional impact at run-time. The layers are outlined in the section 3.1.1.

- **Dependency injection through Blueprint**
  The platform does not only use the OSGi framework to construct its components, it further augments the OSGi bundles using the Blueprint dependency injection framework [5].

3.1.1 Architectural Layers

The OSGi bundles hosted within the architecture platform are divided into three logical architectural layers. The bundles and the services they host that are placed within these different logical architectural layers have different responsibilities and different architectural and design rules applied to them:

- **Presentation layer**
  Bundles in this layer provide services that are consumed by front end applications. Presentation Services are allowed to have session and state dependencies. Presentation services are also allowed to host REST services and to expose their services for consumption by applications outside of the architecture.

- **Integration layer**
  Bundles in the integration layers serve as a middle layer between bundles placed in the presentation layer and components provided by third parties and back-end support systems. Integration bundles can consume REST and SOAP services, generally used to communicate with third party or outside support systems.

- **Business layer**
  This is a bundle that provide services that host some logic or functionality for use by other components. Usually these services are consumed by integration bundles through OSGi dependencies.

3.1.2 Architectural Components

The primary software component that make up the implemented architecture are made up out of a collection of OSGi bundles. The number and makeup of these bundle sub-components are dependent on the functionality the component provides. Two sub-components are mandatory, the implementation bundle and the API bundle. The API bundle defines the interface that is used to consume the services provided by the component, a API is not allowed to have dependencies on other APIs. The implementation bundle implements the functionality outlined in the components API, the implementation may also have dependencies on other components through their APIs. Other types of sub-component bundles are optional and are added as needed. These optional bundles are adapters and models. Adapter bundles are responsible for interacting with third-party components or backend support systems that exists outside of the architecture, adapter bundles are only present in components placed in the integration layer. Model bundles are responsible for modelling and storing the data the component uses. An overview of the outline of a component is shown in figure 3.1 below.

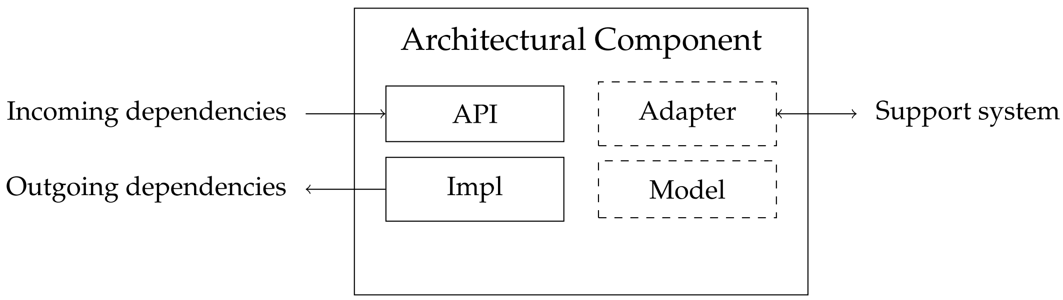

Figure 3.1: Component Representation

4 Method

4.1 Static Architectural Analysis

In order to measure the level of architectural design conformance and structural quality present in the implemented architecture a static analysis method based on the reflexion model [30] was chosen. The model that was constructed in to perform these measurements were also used to measure the level of coupling in the implemented architecture. The architectural quality was gathered from every major release of the company's primary application in order to assess how the architectural quality has its lifetime. The reflexion model was further augmented to better fit analysis of a service oriented architecture by using the measurement metrics used in the LISA architectural design language [54].

4.2 Modeling the Implemented Architecture

In order to assess how well the implemented architecture conforms to its architectural design a modeling tool had to be developed. The tool would have to be able to asses current and past implementations of the architecture in order to evaluate its development. When using the traditional reflexion model [30], this is accomplished either by having developers or architecture manually creating a model based on studying the source code, or alternatively by using code analysis tools to extract an implementation model. While a manual analysis can allow for quicker turnaround for smaller architectures due to the size of the code base, the number of components investigated as well as the scope of time that changes were tracked it is infeasible to model the implementation manually. Therefore, a tool for extracting a model of the architecture implementation was developed. The tool was written in Python 3.5.2, and was designed to gather the available metadata from an archive file containing all the binaries used in a major releases of the company's main application. This is done in order to gather the information needed to construct a model of the implemented architecture. The model of the implemented architecture consists of representations of the actual code modules, called components, that make up the company's backend architecture. The architectural attributes of these components as well as the relationships between them are represented in the model of the implemented architecture. The following sections of this report covers the modeling process of the implemented architecture as well as the quality measurements that was made on it.

4.2.1 Modeling the Components of the Implemented Architecture

In order to create a model of the implemented architecture, each of the software components operating within the architecture was individually modeled. Each of the components dependencies, exposed functionality, and other relevant properties, such as session dependencies, was also gathered in order to determine what architectural layer the component belongs to. Once all the components was modeled and their properties have been extracted from the implemented architecture they were placed in their wider context and in turn the entirety of the implementation was modeled.

Modeling the Bundle

The most common component in a service-oriented architecture based on the OSGi framework is the OSGi bundle. The Jars that make up the OSGi bundles store OSGi specific metadata alongside regular Java metadata in the manifest.mf file that is bundled with any Java Jar archive [2]. This metadata was used to retrieve the following information about the implemented bundle [3]:

- **Bundle Name**
  The name of the bundle that serves as its unique identifier can be extracted from the manifest file.

- **Export Services**
  A OSGi bundle can provide multiple OSGi services. What services that is provided by a bundle is outlined in its manifest file. The tool gathers this data and maps the services to the bundle that provides it.

- **Import Services**
  What OSGi services that a bundle is dependent on. The tool maps this bundle as a consumer of this service and therefor registers the bundle as having a dependency on the bundle providing this service. If a service that is defined as a dependency is not available the bundle will not function.

- **Import Packages**
  What binaries the bundle is dependent on, as well as what versions of those binaries are required. These dependencies also have to be resolved for the bundle to function. This information is used to determine the properties of the bundle and can therefore be used to determine what architectural layer this bundle belongs to.

Since the OSGi container used in the company's architectural platform uses the Blueprint dependency injection framework [5], additional metadata concerning the bundles features could be extracted. This metadata was extracted from the reference file that the Blueprint framework uses. The following additional meteadata was extracted from the Blueprint xml file:

- **Hosting REST or SOAP services**
  In addition to providing services in the form of OSGi services the bundles may provide through REST interfaces. If a bundle provides a REST or SOAP service, it is defined in the Blueprint reference file.

- **Consuming a REST or SOAP service**
  If a bundle is dependent on a REST or SOAP service, the references used to consume it are outlined through services endpoint URLs. This is achieved through specific configuration files provided by the architectural platform.

- **Session dependency**
  In the implementation of the architectural platform, bundles that need access to the user sessions have this functionality injected into the bundle. This injection is outlined in the Blueprint metadata.

By using this gathered metadata, the following component attributes was able to be determined:

- **Session dependency**
  If a component needs access to a user session, it needs to make use of an injectable binary. This session binary is responsible for providing the means to access the user session. Therefore, if the bundle has a dependency on this binary it has a dependency on the user session.

- **Component exposure**
  In order to provide functionality to applications outside of the architecture, endpoints provided by the bundle have to be exposed. Exposed endpoints are then reachable by outside applications, this is mostly done to provide functionality to frontend applications. The functionality that allows the bundles to expose endponts is injected into it through the Blueprint interface. Therefore, endpoint exposure can be extracted from the Blueprint reference file.

- **Component dependencies**
  From the available metadata, a list of what components the bundle has dependencies on can be compiled.

- **Architectural layer**
  What architectural layer a bundle belongs to can be determined by examining the service and binary dependencies that were extracted from available metadata, as well as examining what functionality is injected into a bundle.

4.2.2 The Designed Architecture

In order to assess architectural conformance and other structural violations in the implemented architecture it is compared to an intended architecture, also referred to as the designed architecture. The designed architecture studied in this thesis is not based on a complete pre-planned model of a intended architecture with each component and each component relationship outlined. There does not exist such a model at the company. Instead the designed architecture studied in this thesis is expressed as a set of architectural rules and guidelines. The designed architecture was gathered from architectural rules found in the company's architectural documentation and complemented with information gathered from interviews with the company's software architects. The architectural design that is employed at the company is outlined in greater detail in chapter 3.

4.3 Gathering of Architectural Quality Metrics

The quality of the implemented architecture, once modeled, was measured with three different primary measurements. How well the implemented architecture conforms to the stated designed architecture was measured through consistency analysis. Structural problems were measured through structural analysis and finally the coupling of the implemented architecture were assessed by performing coupling analysis on the dependency information present in the implemented architecture model.

4.3.1 Consistency Analysis (CA)

The designed architecture outlines the rules regarding what relationships a component is permitted to have based on what architectural layer it belongs to. In order to assess how well the implemented architecture follows the designed architecture, violations of relationship rules were recorded. A violation were recoded when a relationship was detected that were not allowed in one or more of the architectural layer the component was determined to belong to. Since a component's intended layer is not outlined in the designed architecture it had to be inferred by the modeling tool from available metadata and component relationships. When a component had been categorized to belonging to multiple architectural layers, any relationship detected that were prohibited in one or more of the layers are counted as a consistency violation.

4.3.2 Structural Analysis (SA)

The structural analysis [54] was performed by applying architectural rules and best practices to the model of the implemented architecture. Some of these rules are based on commonly applied architectural practices, such as detecting cyclic dependencies. While other structural violations are based on design rules that were outlined within the architectural documentation. The design rules that were controlled for are, in what layers components are allowed to expose functionality to frontend applications, as well as how certain third party outside components are allowed to be accessed. The structural architectural rules was expressed programmatically and was applied to the to the model of the implemented architecture. This to gather the numbers of structural problems. The different structural violations checked for are listed below:

1. **Cyclic Dependencies (CD)**
   A cyclic dependency violation is recorded if two components have mutual dependencies, either through direct dependencies or indirect dependencies. These violations are detected by performing checks on the dependencies outlined in the model of the implemented architecture and is measured in the number of cycles detected.
2. **Exposure Violations (EV)**
   Exposing functionality to components outside of the architecture is only allowed for bundles in the presentation layer. Such exposure is therefore counted as a violation for bundles placed in other architectural layers.
3. **Special Component Access violations (SCA)**
   Access to certain third-party and support system components should be regulated by dedicated interface components. It is still possible to access them without these interfaces, but it is against the rules stated in the designed architecture and such access was counted as a violation.

The total of the violations stated above is recorded as SA in the metric table 4.1.

4.3.3 Coupling Analysis (CoA)

The level of coupling between components in the implemented architecture was assessed through collecting measurements on the dependencies between components present in the model of the implemented architecture. The measurements were collected from dependency information, which in turn was collected from the architectural model. Dependencies are not only counted but they are also categorized into one of the following categories [38, 25]:

1. **Direct Dependency (DD)**
   A dependency between a bundle and a OSGi service hosted by another bundle. As well

as dependencies on REST service or a third party components are counted as a direct dependencies [38].

2. **Indirect Dependency (ID)**
Since components will seize to function if their dependencies are not resolved meaning that components indirect dependencies also must be resolved. Indirect dependencies are gathered by iterating over the dependencies gathered by the modeling tool. Long dependency chains are also pointed out to aid in looking for specific structural weaknesses in the architecture.

3. **Dependency on Outside Components (DOC)**
A bundle may have dependencies on third party components or backend support systems through REST or SOAP interfaces. While these components are outside of the control of bundle developers they still need to be resolved in order for the bundle to function. Therefore, these dependencies are weighted equally to regular dependencies. The consumption of these services is collected from extracting what endpoint are used from available metadata and extracting endpoint addresses from source code.

4. **State Dependency (SD)**
When a component is dependent on user sessions it will lead to additional coupling [45]. Therefore, dependencies on user sessions was weighted equally to regular dependencies. State dependencies are collected using the metadata collected from the blueprint file.

In order to properly assess how tightly coupled a component is, it is not only important to measure what dependencies that component have on other components, but also what dependencies those components have. Therefor the following dependencies are also gathered:

5. **Incoming Direct Dependency (IDD)**
The number of dependencies from other components on the component under assessment [38]. This too like DD is collected from the dependence information present in the implemented architecture model.

6. **Incoming Indirect Dependency (IID)**
The number of indirect dependencies that other components have on the component under assessment. This metric is like ID collected from the implemented architecture model.

Each component in the implemented architecture model is given a number representing its impact on coupling, this number is calculated by adding together the aforementioned dependency metrics (*DD, ID, DOC, SD*) and incoming dependencies metrics *(IDD, IID)*, this total is recorded as CoA in the metric table 4.1. The implemented architectures overall coupling is calculated through calculating the sum total of the coupling value of all components in the implemented architecture model:

	CA	CD	EV	SCA	**SA**	DD	ID	DOC	SD	IDD	IID	**CoA**
Bundle A		NA										
...		NA										
Bundle Z		NA										
Architecture Total												

Table 4.1: Architectural Quality Metrics

4.4 Architectural Quality Event Analysis

Larger changes in the architecture's quality over its lifetime was summarized as a list of *events*. This was done in order to narrow down the amount of data gathered and to allow for a more detailed analysis of relevant changes to the architecture and its quality so that this information could be used to select components relevant to architectural quality of the system. These events were created by observing trends in the qualitative data gathered form the architectural model over the available history of the architecture. The architectural components that contributed to these events were used as a basis for selecting relevant developers as interview subjects to gather additional insight as to the causes of these changes in the quality of the architecture, see section 4.6 below for more information on the interviews and section 5.4 the results chapter for more information on the events.

4.5 Internal Validation of Modeling Software

In order to verify that the modeling tool developed during this study, as outline in section 4.2. As well as any measurement preformed using the models, as outlined in section 4.3, could be relied upon, an internal validation of the modeling tool was performed.

4.5.1 Component Representation Validation

In order to ensure that a model of an implemented architecture represents the actual implementation the components present in the model was compared to their source code counterparts. The following attributes was validated during the validation:

- **Layer placement**
  The criteria used to place a component in a certain layer was validated by selecting components present in the model and confirming with the developers responsible for these components.
- **Endpoint usage**
  The dependencies of outside components were verified by confirming those dependencies through looking up what endpoint URLs are used in source code and outlined in configuration files that is used in the implemented architecture.
- **Incoming and Outgoing Dependencies**
  Regular dependencies between OSGi components were verified through manual checks on stated dependencies in metadata and source code.
- **Other component properties**
  Other component properties such as session dependencies, integration interfaces, component exposure and usage of special components were verified through checks on source code components as well as verification with the responsible developers and architects.

4.5.2 Structural Analysis Validation

Once components were verified, the data extracted from the overall structure of these components also had to be verified. Any detected cyclic dependencies, exposure violation or Special access component violations also had to be validated:

- **Cyclic dependencies**
  Cyclic dependencies was tracked and examined in the source code to validate that they were genuine.

- **Exposure violations**
  The verification of exposure validations was performed by examining the criteria the analysis tool used to conclude that a component has improperly exposed service, the components was then manually examined to determine if those criteria applied.

- **Special access component violations**
  The usage of special components was verified through checking the usage of these components in the source code.

4.6 Interviews

A interview-based data collection methodology [40] was chosen to gather insight into the causes of the changes in architectural quality metrics detected in the aforementioned events, see section 4.4. A series of semi-structured interviews were conducted with different developers from different development teams. The interview subjects were selected to provide as a broad selection of developer's perspectives as possible. The developers were selected with aspect to how long they have worked with the architecture, what kind of components the developers worked on within the architecture and if they were responsible for developing back-end, front-end or both kind of components. All the subjects are a part of development teams that had worked on components that were a part of the changes in the aforementioned events.

4.6.1 Interview Structure

The interviews were conducted in the following six phases. Each phase contained a few questions regarding a particular topic and were asked in the order was most appropriate to the discussion, the interviewer also asked follow-up questions to gain additional information where appropriate.

1. **Introduction to study**
   The purpose and structure of the interview as well as the purpose and goals of the thesis work was presented to the interview subject in this phase. The consent of the interview subject to perform the interview and use the results in the thesis is acquired here as well.

2. **Background of the interview subject**
   A few questions were asked about the subject's background to give him/her an opportunity for the subject to introduce themselves, what position they had at the company and within their development team and how long they had worked within the architecture. These questions were asked to put the interview subject's insights into prescriptive as well as making sure that the interview subjects used for the study represent as a wide set of perspectives as possible.

3. **Architectural considerations during development**
   General questions about the developers experience when developing new or updating existing components within the architecture. The questions were directed towards covering the different measurement metrics as covered in section 4.3.

4. **Architectural maintenance tasks**
   Questions about tasks of maintaining and improving architectural quality as well as questions regarding refactoring of existing components were asked during this phase. The questions aimed at understanding if the subjects team priorities when it comes to maintenance or refactoring of components and the overall architectural quality. Questions about the knowledge of components in need for maintenance or refactoring and

what effects the subject believe that this work would have on the component and the architecture.

5. **Specific Component investigation**
Specific questions about the development of the components that were a part of the architectural events that lead to the interview subject's inclusion in the study. Other components of interests that were within the interview subject's team's responsibility were also included. The questions asked during this phase were designed to gather more insight to the reasoning of the design of these components in order to understand why they are constructed the way they are.

6. **Interview wrap up**
The interview is ended with a wrap up were the interviewer presents a summary of the interviews findings. The subject is also asked if they are available for follow up interviews if needed.

The interviews were recorded and notes were taken throughout, and later were transcribed and summarized. A full protocol of all questions asked cannot be provided in this report due to concerns of confidentiality. However, a broad overview of the types of questions asked of the questions asked are outline in the interview protocol found in appendix A.1. Once the interviews were transcribed and summarized, the summery were sent back to the interview subject for comment and review.

4.6.2 Conclusions Based on Interview Data

The methodology used for analyzing gathered quantitative interview data and drawing conclusions from it as described in *Guidelines for conducting and reporting case study research in software engineering* by Runeson and Höst [40] is listed and outlined below:

1. **The architecture and its development**
The design and the further development of the software architecture is the subject under study. Working with this architecture gives the developers a perspective on its quality and development.

2. **The developer's perspective**
The developers that develop new and update existing components within the architecture gathers observations and experiences of its quality. These perspectives are gathers through performing interviews with these developers.

3. **Sound recordings of the interviews**
In order ease the process of gathering data from interviews the interviews were recorded as an audio file.

4. **Transcription of sound recordings**
To provide a more easily used source for the gathering of quotes and analysis material the recorded interviews were transcribed.

5. **Grouping of quotes**
During the transcription process as well as the subsequent analysis of the transcriptions relevant quotes were gathered. The quotes are relevant to the topics under study and are mapped to the interview subject's properties, like developer role and time working within the architecture. The quotes are later grouped together with other quotes that regard the same topic given by other interview subjects.

6. **Drawing conclusions**
The grouped quotes were used to draw conclusions about the state of the architecture,

as well as the reasons that contributed to this state. The conclusions are later matched to other conclusions in theory and related works to support or discredit its validity.

4.6.3 Interview Subjects

Interview subjects were selected from the different development teams that had been involved in the changes that contributed to the events of major changes in measured quality metrics. More about the events of major change can be read in section 5.4 in the results chapter and the events themselves are outlined in table 5.1. The majority of the developers interviewed were backend developer's due to the fact that the implemented architecture that is under study provides backend functionality to the company's web-applications. A single frontend developer were also interviewed and two fullstack developers. A fullstack developer is a developer that both develops frontend and backend components. The developers were also selected for the amount of time they had been working at the company, the subjects were divided into three time spans, short, medium and long. Were short is less than two years, medium is between two and five years, and long is five years or more of working experience within the architecture. A breakdown of each interview subject is outlined in table 4.2.

	Developer role			Architectural experience		
Subject number	Frontend	Fullstack	Backend	Short	Medium	Long
1	✓				✓	
2			✓	✓		
3			✓		✓	
4			✓			✓
5		✓		✓		
6			✓	✓		
7			✓		✓	
8			✓	✓		
9		✓				✓
10			✓		✓	
11			✓			✓
12			✓			✓
Total	1	2	9	4	4	4

Table 4.2: Selection of interview subjects

4.7 Architectural Reengineering

The model of the implemented architecture was used to reason about, measure, evaluate and model planned changes to the architecture with accordance to the to the *Analyze* and *Restructure* steps of the *Horseshoe Model* method [17]. This approach was partly chosen due to the fact that an architectural model already had been constructed. But as the method allowed the thesis writer to perform architectural reengineering with little input from architects. This was possible due to the fact that the planned architectural migration had not intended to change the functional requirements of the architecture, only to better fulfill the already existing ones. If new architectural requirements had to be fulfilled then a different method would have been more appropriate, like the scenario-based architectural reengineering methods[10] that outlines methods for architects to reason how to change existing architectures to fulfill new requirements.

This microservice reengineering was conducted in two major steps. First, planned changes to the architectural platform itself was modeled. Second, a few select components and component clusters were reengineered as microservice components and was introduced in to the model.

4.7.1 Visualization of the Software Architecture

In order to ease the process of selecting components for reengineering and to reason about the architectural model was visualized as a graph. In the graph, each component was represented as a node with its OSGi dependencies to other components represented as the edges in the graph. The graph was based on the same model used to measure architectural quality metrics 4.2 and was constructed using the GraphML graph file-format [8] and was visualized and organized using the yEd graph editor [55].

4.7.2 Reengineering for Planned Changes in the Platform

The new planned architectural platform had been designed to adopt certain vital component into to the underlying platform, the functionality that these components provide will be provided through the underlying platform through injection and warping of components and calls to third party systems. The first step in the reengineering process was therefore to remove these components from the model and recalculate the qualitative metrics.

4.7.3 Microservice Reengineering

In order to assess the impacts of the planned gradual reengineering towards a microservice majority architecture, components and component clusters present in the architecture were selected to be reengineered into microservice components within the model of the implemented architecture. Once candidates had been selected and modelled within the redesigned architecture, there impact of these redesigned components were measured using the same metrics as outlined in section 4.3 with two additional coupling metrics, *Web service dependencies (WD)* and *Microservice to microservice dependency (MD)*. The same measuring methods, as outlined in section 4.2, was used with the addition of the extra metrics and some removed measurement metrics as outlined in the section *Removed measurements* below.

Web service dependencies (WD)

Since the microservice components are not based on OSGi they cannot access the functionality of OSGi components through the regular dependencies that were outlined in section 4.3. Instead, a microservice component that have a need to consume some functionality provided by a not yet migrated OSGi component must do this through a REST interface. This means that for this web service dependency to be satisfied the OSGi component must be updated to provide the functionality normally provided through OSGi through a new REST hosting interface.

Microservice to microservice dependency (MD)

A microservice to microservice dependency function in the same way as a web service dependency, the only difference is the component hosting the service.

Candidates for Microserviceization

Components present in the model of the implemented architecture were analyzed to find candidates for being reengineered into microservice components. The criteria used for identifying microservice candidates are listed below and are based on the microservice design recommendations outlined in the book written by Neuman [31] and the paper by Escobar et al [12] with some delimitation.

Removed measurements and assumptions

As the microservice components differ from the OSGi components in the ways they handle dependencies. A OSGi dependency is hard and a component will not function without having the it's dependency fulfilled. This is the primary reason for including indirect dependencies as a coupling metric, while microservice component make use of web interfaces to handle dependencies on other components. These dependencies do not necessarily need to be fulfilled for the component to function depending on the implementation, therefore indirect dependencies are not counted when the dependency originate from a microservice component. Another assumption is that any migrated component that have any dependencies on outside components or have to preserve user states are carried over and that any violations related to these properties are resolved during the migration. These properties are therefore not measured when the migrated architecture is analyzed but they are instead assumed to be the same.

Candidate Criteria

- **Independent components**
  Component with no or very few OSGi dependencies components are obvious candidates for easy being reengineered into microservice components due to there already high level of loose coupling. If the components already use REST interfaces the transformation is further trivialized. It is however also important to note that the increase of architectural qualities.

- **Component Clusters**
  Components that through their dependencies cluster together as well as share the same domain. Clusters were identified manually using the generated graph. The components belonging to the cluster candidate were confirmed by architects and developers that they belong to the same domain. Due to the nature of the planned microservice enabled platform that will be used at the company a migrated component cannot provide any functionality to a none-migrated component. Meaning that any cluster selected as a migration candidate may not have any incoming dependencies from any OSGi bundles. An example of such a transformation is show in figure 4.1.

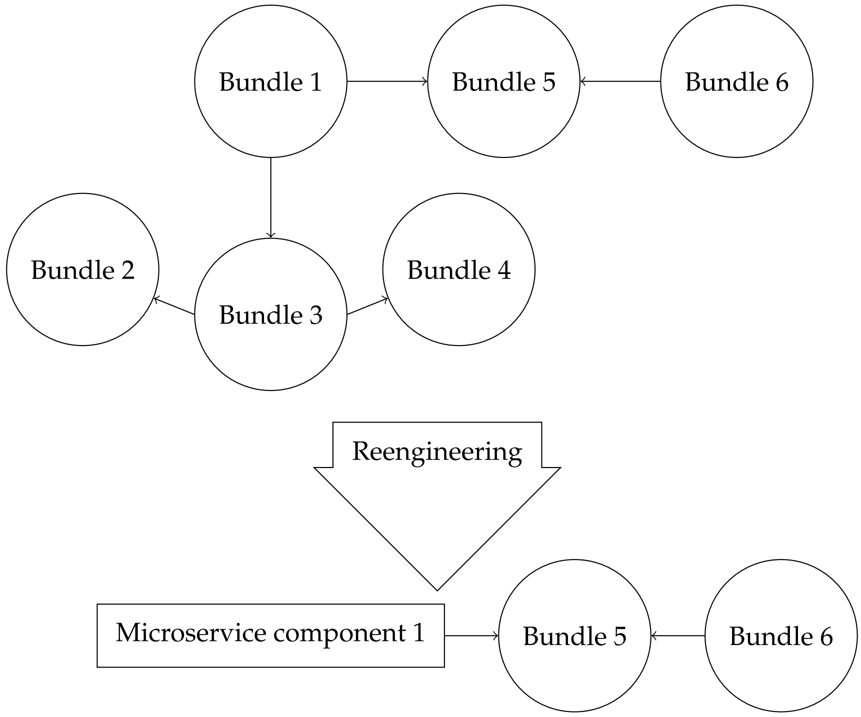

Figure 4.1: Example of Cluster to Microservice Reengineering.

- **Clusters of microservice candidates**
  Due to the dependency constraints between OSGi bundles and microservice components mentioned above some clusters have to be reengineered together. This creates multiple new microservice components and introduce a dependency between them. This type of reengineering was discouraged by architects at the company since it will introduce dependencies between microservices, and thereby go against the goal of keeping microservices as independent as possible. An example of such reengineering can be seen in figure 4.2.

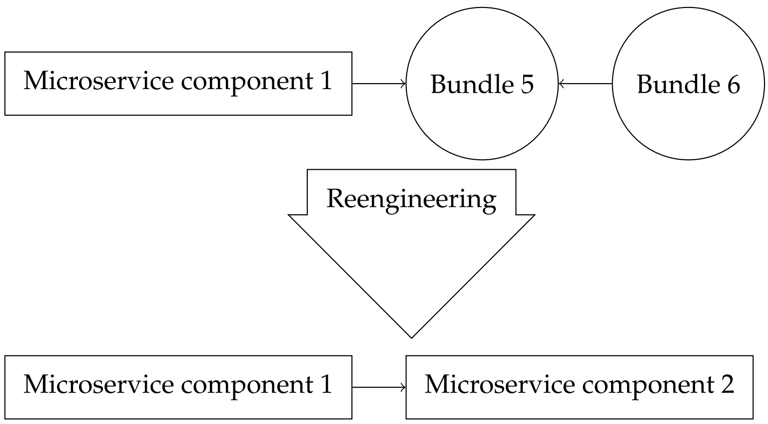

Figure 4.2: Example of Microservice to Cluster Microservice reengineering.

Candidate delimitation

As mentioned above, some delimitation had to be accepted when estimating what a microservice migration would look like and what effects it will have on the overall architectural quality.

- **No component duplication**
  While almost all instances code sharing in the current implemented architecture are handled through OSGi dependencies, this might not be the best option for certain migrated components. For instance, if two component clusters share a dependency to the same business layer component, it might make sense to copy this component so that it is a part of both microservice components when migrated. However, to make such a decision requires a precise knowledge of the domain as well as the components to be migrated. This is outside the scope of this thesis. Therefor no component duplication is performed during the reengineering process.

- **No component removal or addition**
  While the addition or removal of functionality are commonly performed during a reengineering process no such transformations was performed during the reengineering done in this project. This delimitation exists because such actions would require greater understanding of the components and their domain than was possible to acquire during this thesis project.

- **Dependencies are the primary indicator of domain allegiance**
  The primary indicator used to determine whether a component would be appropriate to cluster with another to create a microservice candidate. While some microservice candidates was selected due to some domain knowledge outside of the components dependencies the vast majority were not.

- **Microservice complexity**
  No consideration to the internal structure or complexity of a proposed microservice candidate were considered when selecting OSGi components for microservice reengineering.

A clustering algorithm as outlined by Escobar et al [12] were considered but was rejected. While automating this process would generate more possible clusters candidates for microserviceization the algorithm would still have to be implemented and would not be able to take any other domain information into account other than dependencies. More about this in the related work section, section 4.8 below.

Reengineering steps

The microservice reengineering was performed in three steps, were each of the steps used slightly different criteria for the cluster selection.

1. Clusters of OSGi components that represent the far edges of the dependency graph were selected. No microservice to microservice dependencies were allowed.

2. Further OSGi clusters were selected. Some microservice to microservice where allowed but highly coupled components were avoided to limit the microservice to microservice dependencies.

3. All remaining components, including single components that are not part of a cluster were converted into microservices.

4.8 Related Work

This section serves to present related works to give context to the methodology used in this study. The topics of microservice migration, the study of evolution and erosion of software architectures and static architectural analysis will be covered.

While the microservice architectural is fairly new there exists, studies examining method's and results of implementing them. Finding papers that presents any methodology that could be applied to migrate SOA architectures, however a notable exception to this is the paper "Towards the Understanding and Evolution of Monolithic Applications as Microservices" written by Escobar, Daniel, et al [12]. In the case study presented in the report a coupling based clustering algorithm was used to visualize and migrate a monolithic architecture towards a microservice based architecture. The implemented architecture studied in the paper is similar to the architecture studied in this thesis as is a Java based, three layered, web application backend architecture. The architecture that is studied by Escobar is a layered monolith rather than a service-based architecture that is studied in this thesis project. The clustering algorithm detects elements within the original architecture using dependencies and inheritance. The algorithm also calculates how related in dependencies and invocations between these clusters to determine if these clusters should be merged. The study experiments with different levels of coupling required between clusters of components and therefore the level of cohesion within the clusters to be considered candidate for microservice reengineering. The papers algorithm exports its results as a visualized UML diagram that could be used during the architectural transformation process. The authors point out that this automated process cannot be used without architect input but instead be used to help with the migration process. While this study presents an algorithm-based approach similar to the manual clustering methodology used in this study. However, it does not attempt to assess the changes in architectural quality that such a reegenginering effort would have on the architectures quality. Nor does it examine how the current architecture that is being transformed has changed during its lifetime. Something that this thesis project attempts to do.

In the paper written by Van Gurp, Jilles, Sjaak Brinkkemper, and Jan Bosch [50] presents the results of two different case studies conducted on the topic of architectural design erosion during software evolution. The two case studies studied the evolution of two different systems. The first was database query processing subsystem that have been evolved over time to support an increasing amount of query languages and databases. The second case study studied a purchase and sales schedules, that have been evolved over time to satisfy the ever-changing requirements of new customers. The research questions were concerning identifying the *Symptoms* of design erosion. The *Identification* of eroded systems and what needs to be done to fix them. The *Causes* for software erosion in systems and how to *Resolve* known erosion issues. And finally, how to *Prevent* architectural design erosion during evolution. The studies were primarily of a qualitative nature using developer interviews, but these interviews were complemented with quantitative data that is some cases were gathered from the interviews, such as estimates of defects during development, but also data gathered from documentation and source code analysis. To further increase the validity of the conclusions drawn in the paper multiple independent developer interviews were conducted allowing for cross checking of interview results and other relevant quantitative data. The interviewers also requested that the interview subjects provide feedback on the results and conclusions to minimize the probability of misunderstandings. A flaw with this study that is brought up by its authors is its overreliance on qualitative data. While quantitative data was used a complete architectural analysis was never performed, nor was not data tracked over time other than developer reported bug and issues. This thesis attempts to combine the qualitative data gathering method employed in Van Gurp, Jilles, Sjaak Brinkkemper, and Jan Boschs study with a static analysis method to back up the quantitative conclusions with relevant quantitative gathered from the architectures lifetime.

Another similar approach to study architectural erosion during is evolution is the ones taken in the paper written by Macia, Isela, et al [28] and the paper written by Jaktman, Catherine Blake, John Leaney, and Ming Liu [24]. In these papers, a more quantitative approach were used to find if there were any relationship between different code quality and architectural structural metrics and the level of design erosion. In the later paper architectural complexity metrics such as the level of coupling through dependencies and function call paths are tracked alongside the level of design erosion as observed by developers that had been reported during interviews. The case study was able to map some characteristics of design erosion, such as the increasing of architectural complexity per release, the increase of unforeseen behavior as results of implemented changes and the mismatches between planned architecture and implemented architecture, to the results of their static analysis and further validated by the developer interviews they performed. The other paper uses a similar quantitative approach were a static code analysis to detect code anomalies to see if they can be used as indicator of architectural erosion. The level of architectural erosion was measured using the reflexion model and once cases of were detected the amounts of code anomalies that lead up to its creation was investigated. The paper concluded that a majority of the detected cases of architectural erosion could be linked to different measured code anomalies.

5 Results

This chapter serves to present the results of the study. In section 5.1 the implementation of the static analysis tool is outlined. In section 5.2, presents an internal validation of the tool developed to give further confidence in the results presented in section 5.3. In that section the results of the static architectural analysis are presented, these results serve as the basis for the conclusions reached regarding the first and second research question of this thesis. In section 5.5 the results of the developer interviews are presented. These results serve to give further basis for the conclusion drawn regarding the second research question. Finally, in section 5.6 the results of the Microservice Reengineering process are presented, that serves as the basis for the conclusions regarding the third and final research question.

5.1 Implementation of the Static Architectural Analysis Tool

In this thesis project the choice of developing a custom tool for modeling and measuring the quality of the architecture was made fairly early. This was due to the fact that no tool that could be used "out of the box" could be found that also gather the necessary metrics and analysis. The development of the was developed solely by the thesis writer on and off over two months. However, the thesis writer did not work on it full time and most of the was devoted into understanding the architecture. If the tool had been developed by a more experienced developer with more domain knowledge it could have been completed considerably faster.

5.1.1 Running the Analysis Tool

To run the tool a user must provide a zip file containing all the components making up the architecture as well as some configuration metadata files. Modeling and analyzing one version of the company's backend architecture takes around five minutes running on a developer workstation machine. This indicates that this tool or a similar tool can reasonable be used to analyze much bigger architectures without any major performance issues.

5.1.2 Overview of the Tool

As motioned several times in this report, the full source code of the tool can not be provided as this risk harming the integrity of the company's software. However, some detail can be revealed in this section. Pseudo code is not provided, as most of the code in the tool are simple loops scanning files for certain strings. But a general step by step description of the modeling tool is provided below.

1. A zip-archive containing every OSGi-bundle making up the backend architecture is used as input.
2. Iterate over every .jar file proved in the archive.
   - 2.1. The bundle type, API, Impl, Adapter or model is determined trough meta data found in the .jar archive.
   - 2.2. Bundles belonging to the same architectural components are identified through the name of the .jar archive. Bundles belonging to the same component have the same prefix as part of their name.
3. Each of the identified components implementation bundle is iterated upon, where the java manifest file and blueprint dependency injection files are extracted.
   - 3.1. Bundle name and version is extracted from manifest.
   - 3.2. Dependencies on OSGi services as well as what OSGi services are provided by the bundle is extracted from the manifest.
   - 3.3. Session dependencies are extracted from blueprint file.
   - 3.4. REST or SOAP dependencies are extracted from blueprint file.
   - 3.5. Each java class contained in the implementation bundle is scanned for URLs used as endpoint dependencies. This is used to determine dependencies on REST service.
   - 3.6. The maven pom file is analyzed for the usage of wsdlFiles.
4. The api-bundle belonging to the component analyzed to determine the following.
   - 4.1. If the bundle is providing any services to the presentation layer.
   - 4.2. If it has integrations to the company's third-party components.
5. What layers the component belongs to is determined through a set of tests performed on the components meta data. The tests cannot be reviled due to integrity issues.
6. Once every component has been model, they are iterated upon again to.
   - 6.1. Map every service dependency to the component that is providing that dependency, creating a graph where each component is a node and each dependency is an edge
   - 6.2. Calculating coupling as the graph is being created.
   - 6.3. Reporting any cyclic dependencies detected while constructing the graph.
   - 6.4. Reports any consistency violations in accordance to the components layer.
   - 6.5. Reports structural violations detected by reporting exposure violations as well as any incorrectly handled connections to third party components as special access violations.
7. The architectural total is calculated and saved. The graph is exported as a graphml file.

5.2 Internal Validation of Modeling Software

As mentioned in section 4.5 an internal validation was performed in order to verify that the tool that was developed to perform the modeling and the static analysis of the architecture provided an accurate model of the implemented architecture as well as an accurate assessment of its architectural quality.

5.2.1 Component Representation Validation

The primary method for validating whether or not the architectural model provided any false positives or missed any details. Manual checks were performed by the thesis writer. Not every component were validated due to the number of components of the architecture. A subset of components were selected for their representation of different component types present in the architecture. The number of validated components equaled 120 of the total of 231 components modeled in the architectural model. When a discrepancy between the model of the implemented architecture and the source code was detected it was investigated and solved, once resolved the discrepancy was re-verified.

5.2.2 Structural Analysis Validation

Once the components had been verified the data extracted from the overall structure of these components were also verified.

- **Cyclic dependencies**
  Each of the cyclic dependency that were detected was caused by components consuming service beans and then by mistake exposing those beans later. By doing this the bundle have in effect created a dependency on itself and thereby a cyclic dependency have been created within the metadata.

- **Exposure violations**
  When the exposure violations were investigated no false positive results were detected. Also, out of the components with no reported instances of exposure violations that were investigated no previously undetected exposure violations were discovered.

- **Special access component violations**
  A selection of the dependencies that were marked as violating the special component rules were verified that they did not use the appropriate interface components to securely access these components.

This validation process verified that a majority of the components were present in the model of the architecture but it did not cover all components in the model. However, a few model anomalies were detected and still exist within the model. A model anomaly is a component within the architectural model that misrepresents the actual implemented component, due to missing information or erroneously interpreted metadata. These anomalous components are special cases, often due to the nature they are generated not providing sufficient metadata to properly model them, 15 of the total 231 were detected as being anomalous. Seven of these anomalous components were identified as being mismatched as independent components while they actually are sub-component to other components within the architecture. One component was identified as being deprecated and had been fully replaced with a newer one with the same name, while the rest are generated or used in none standard ways so that the methods for modelling them fail to properly represent their impact on the architecture.

5.2.3 The discrepancy between incoming and outgoing dependencies

During the validation of the coupling measurements as shown in the results table A2T1 found in appendix A.2 a discrepancy between the number of incoming and outgoing dependences were detected. The number of outgoing OSGi dependencies, called **direct dependencies (DD)**, are greater than the number of incoming OSGi dependencies, called **incoming direct dependencies (IDD)**. This would intuitively indicate a measurement error, however this discrepancy has been investigated and it has been determined that this discrepancy is due to how certain components express dependencies on platform functionality or outside components through a OSGi dependency. It could be argued that these dependencies should therefore be classified as **dependencies on outside components (DOC)** for a less misleading result. However, during the development of the measurement software, a method was introduced for determining if a measured OSGi dependency was to an architectural component or a platform component that did not require manual checks towards known platform functionality.

5.3 Static Architectural Analysis

In order to perform the static architectural analysis, the modeling tool was run on the 57 previously released versions of the implemented architecture, these versions of the implemented architecture date back over one year of continuous development and releases. The total number of components in the implemented architecture is plotted in figure 5.1, the number of consistency violations are plotted in figure 5.2, the number of structural violations are plotted in figure 5.3, and finally the value of the coupling metrics is plotted in figure 5.4. In table A2T1 the data used to plot these graphs and it can be found in appendix A.2.

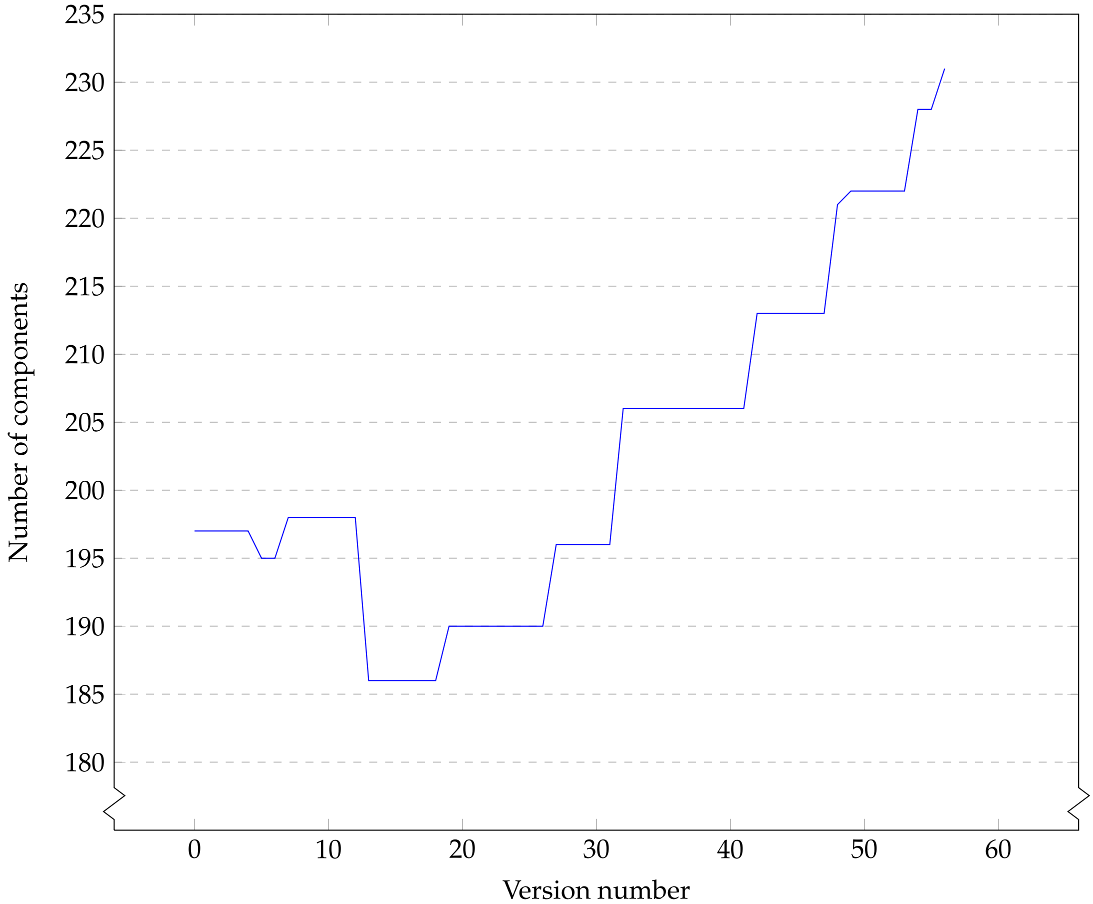

Figure 5.1: Number of Components

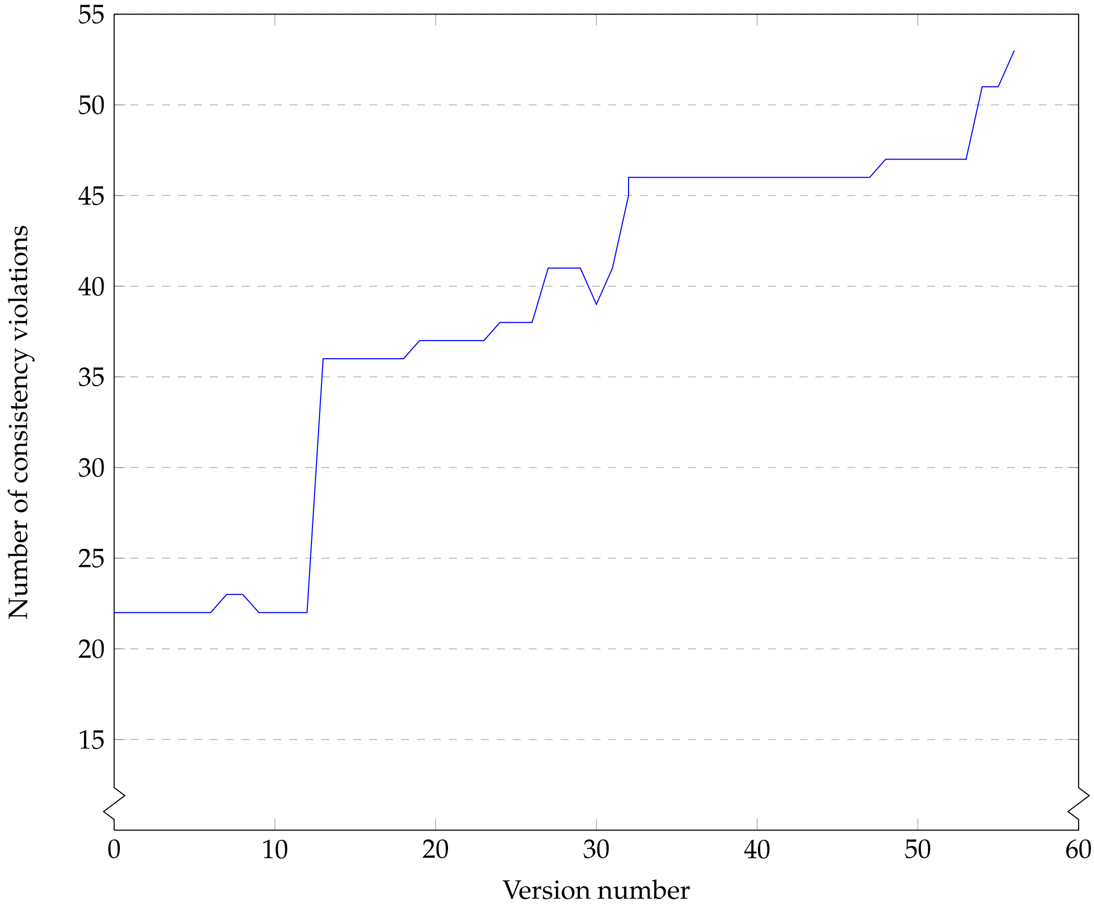

Figure 5.2: Number of Consistency Violations

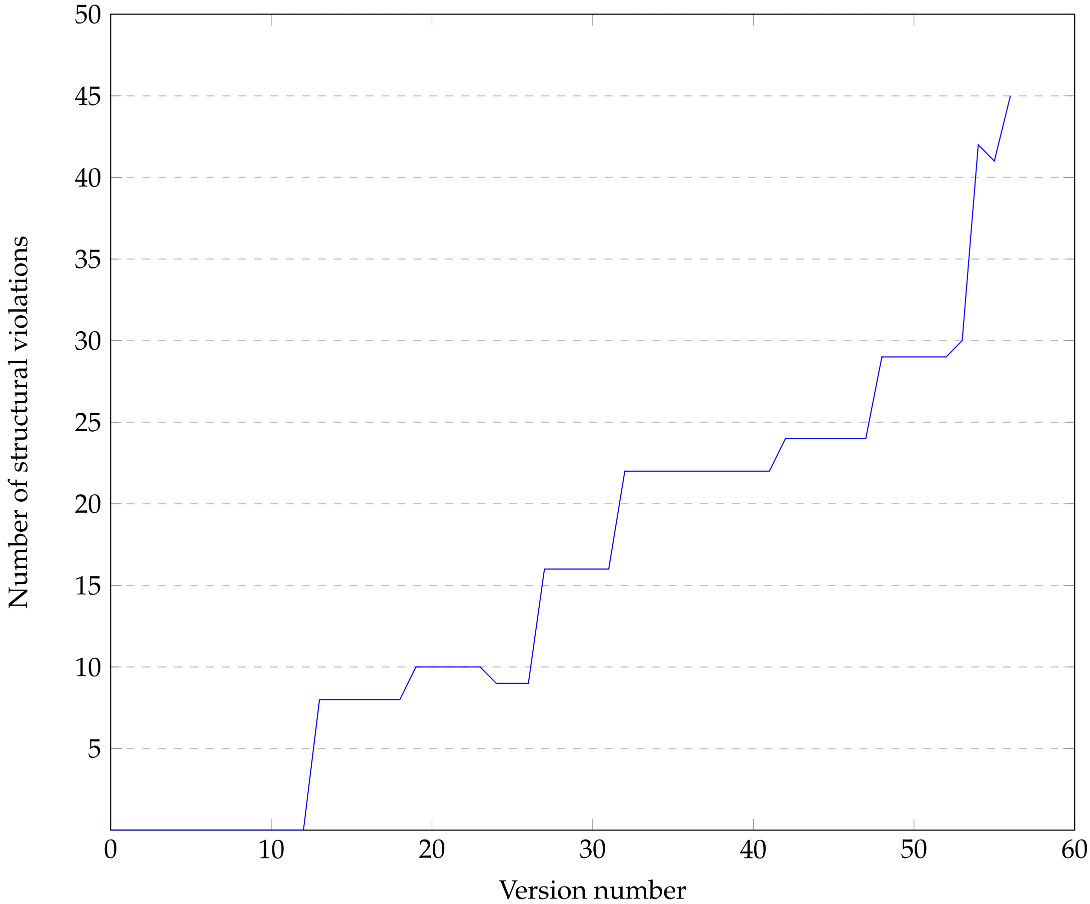

Figure 5.3: Number of Structural Violations

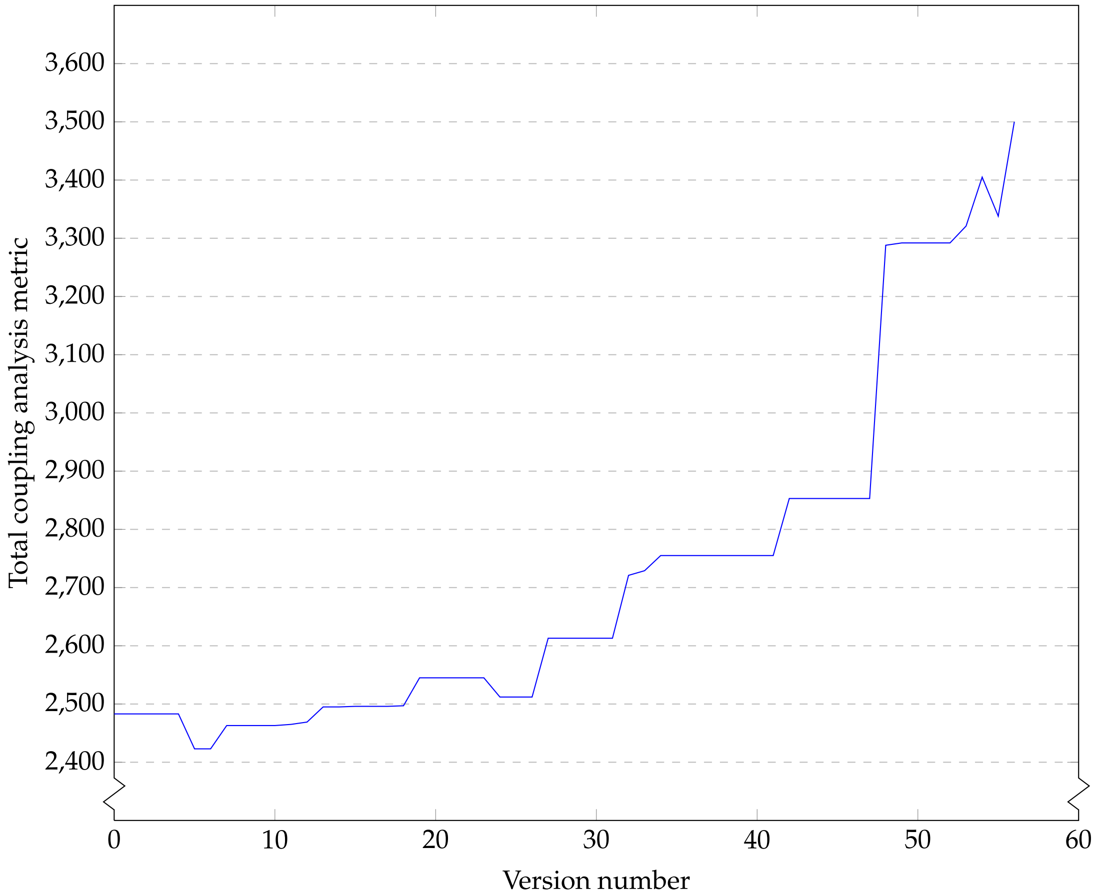

Figure 5.4: Coupling Analysis

5.4 Events of Major Changes in Qualitative Metrics

In order to narrow down the amount of data to be analyzed and to allow for more focus on relevant trends a few events were chosen from the data gathered from the model of the implemented architecture. These events were selected as they represent the largest consecutive changes in measured quality metrics. The events and the changes in architectural quality are listed in table 5.1. If multiple consecutive versions indicate a trend in the changes of architectural quality attributes they are grouped into the same events, therefor the *version span* column indicate the consecutive version of the architecture included in the event. The following metrics in the table are expressed as the difference of these metrics between the first and last version of the architecture within the event span.

Version span	Δ Bundles	Δ CA	Δ CD	Δ EV	Δ SCA	Δ **SA**	Δ DD	Δ ID	Δ DOC	Δ SD	Δ IDD	Δ IID	Δ **CoA**
12 → 13	-12	14	1	0	7	8	-36	9	52	-1	-7	9	26
30 → 33	10	7	0	4	2	6	20	37	4	0	19	36	116
41 → 42	7	0	0	2	0	2	6	39	3	0	11	39	98
47 → 48	8	1	0	3	2	5	23	155	5	0	97	155	435
53 → 56	9	6	0	7	8	15	27	58	4	3	29	58	179

Table 5.1: Events of major changes in quality metrics

5.5 Interview Conclusions

This section presents the conclusions that were drawn from the developer interviews. The conclusions will be listed by topic and be presented alongside relevant context and summaries of relevant quotes gathered from the interviews. The selected quotes grouped are presented alongside the summaries presented in the following sections will refer to a specific grouping of quotes that can be found in the following sections. The quotes have been translated from Swedish and some parts like names, gender pronouns, or other sensitive information regarding the architecture, the platform and software have been altered in order to preserve the integrity of the interview subject and the software that is being discussed. The altered segments have been altered in ways to preserve their context and have therefore been replaced with context appropriate keywords surrounded by single quotation marks. For example, a quote containing a name of a developer might be written as follows.

> "I asked my colleague 'Name' for help but he/she did not know how to do it either."

In Figure 5.5 lists summaries of the conclusions drawn from the interviews. The quotes that were used as the basis for these conclusions are outlined in the following sections, **Architectural familiarity**, **Coupling and architectural considerations during development**, **Maintenance and refactoring of components**, **Layer placement related violations** and **Development methodology and the release cycle** that all follow this section in the report.

C1 Architectural documentation is almost only used when the developers first is familiarizing themselves with the architecture but almost never after that.

C2 Architectural knowledge is primarily acquired through colleagues and existing code within the same development team.

C3 The number of new developers have increased and few senior developers have remained since the introduction of the architecture.

C4 Some Developers are uncertain of the intended architectural design when developers make design decisions during development.

C5 The developers are uncertain if they should prioritize code reuse or low coupling.

C6 The architectural impact of dependencies is often not considered during development.

C7 Dependencies between components contribute to difficulties when updating or removing these components. Often resulting in the duplication of components to facilitate migration, further increasing architectural complexity.

C8 Dependencies between components belonging to different teams introduce further difficulties and complexity to the architecture.

C9 Components with uncertain ownership add additional complexity and are not maintained.

C10 Components with a short planned lifetime are rushed and often have a longer lifetime than planned.

C11 Maintenance work regarding known issues are often not prioritized unless they are combined with new requirements or fixing critical issues.

C12 Developers mostly perform maintenance and refactoring work on components when they have been assigned a none maintenance related task related to the components.

C13 Inadequate tests and test data prevents developers from performing maintenance and refactorization of components.

C14 The layer structure has largely been abandoned by many of the development teams.

C15 The primary reasons for increase in special component access violations is the lack of knowledge of the interface functionality provided by the architectural platform.

C16 Difficulties with properly configuring the interface functionality is a contributing factor to why developers choose not to use it.

C17 The release cycle contribute to the rushed development of components.

C18 Rushed components often have to immediately undergo maintenance after release.

Figure 5.5: List of Conclusions Drawn from the Interviews

5.5.1 Architectural Familiarity

To further understand the relationship the developers had with the architecture, a few questions were asked as pertaining to how experienced they are at working within it. The questions also covered the developers experience of learning the architectures design rules and the functionality provided by the architecture. When discussing the topic of the state and usage of the architectural documentation an obvious pattern emerged, if the architectural documentation was used at all it was used when the developers were first familiarizing themselves with the architecture, but the documentation was rarely refereed to after that. The quotes outlined in figure 5.6 seen below serves as the basis for the first of the interview conclusions, conclusion C1: *Architectural documentation is almost only used when the developers first is familiarizing themselves with the architecture but almost never after that.*

From interview 3:
Interviewer: "What did you do to familiarize yourself with the architecture when you started working here? Did you read the documentation or did you mostly work and ask for help when you ran into problems?"
Subject 3: "Yes, exactly, I read the documentation fairly little. I mostly used it to get everything running on my computer."
From interview 6:
Interviewer: "Have you read the documentation, about the rules and recommendation regarding the architecture?"
Subject 6: "When I began working here there where some of that, a bit about how the architecture was constructed and how we should work."
Interviewer: "Is this something you have referred to later or was it something you only used when you began working here?"
Subject 6: "It was only when I began here."
From interview 7:
Interviewer: "How did you familiarize yourself with the architecture and platform when you started working here?"
Subject 7: "First I worked with the developer handbook, then I worked through legacy code."
Interviewer: "Do you refer back to the documentation when you work?"
Subject 7: "No, not really."

Figure 5.6: Architectural documentation

However, a far more common way for the developers to familiarize themselves with the architecture is by being mentored through other team members or platform developers. Another common alternative to reading the documentation is through the study of existing code that is owned and maintained by the developer's team. A grouping of quotes related to these are shown in figure 5.7. During the interviews, interview subject 2 did also mention that the primary way he/she familiarize themselves with the architecture was to work with existing components. However, that this way of learning architectural design did leave many gaps in the understanding of the overall architecture. And when asked if he/she and their team strived to follow the rules and design when working within it, he/she responded that they did try. However, that it is hard to know what the 'rules' are and to gain a design overview of the current state of the architecture, especially now when they work as autonomous teams. When furthered asked how the documentation is used during development he/she said that it is not used, and that he/she did not see a need to use it. The quotes and observations just states served as the basis for the conclusion C2: *Architectural knowledge is primarily acquired through colleagues and existing code within the same development team.*

From interview 5:
Interviewer: "How have you gathered your knowledge of the architecture? Did you study the documentation or have you mostly talked with other developers?"
Subject 5: "When I began working here I got a mentor form my team, 'Mentor name'. He/She did explain how everything worked in broad strokes, and later how you make use of it. So, I have mostly asked him/her, and I think they have managed to set it up in a good way, it is easy to understand. But I mostly ask, 'Mentor name' or the DevOps Team or the platform developers."
From interview 10:
Interviewer: "How was it to begin working within the architecture, how did you familiarize yourself with it? Did you read documentation or did you mostly study existing code?"
Subject 10: "There did not exist much documentation to work with back then, I mostly studied existing code and asked questions."
From interview 8:
Interviewer: "When you began working within the architecture, how did you begin to familiarize yourself with the architecture? Did you read the documentation or did you began working and just learned as you went?"
Subject 8: "Docuwhat? No, I read and studied the code, I am fairly experienced with reading code and understand what it does and suggesting improvements. Essentially performing code reviews, and that was how I learned."
Interviewer: "Did you get any intuition for the architectures rules and design by working with the code this way?"
Subject 8: "No, not a lot. Sometimes I have to talk to our team architect and ask questions regarding why it is designed like this."

Figure 5.7: Architectural Mentoring

5.5.2 Coupling and Architectural Considerations During Development

As seen in table 5.1 and in figure 5.4 the total coupling complexity have been increasing over almost the entire measured lifetime of the architecture. This increase in complexity seems to correlate to, but not entirely connected to the introduction of new components in the architecture. To further understand this increase in coupling complexity as well as the increase in the number of components within the architecture a few questions regarding the development of components and the dependencies between them during phase 3 of the interview. Future questions about particular components of interest was also asked during phase 5 of the interview. When the interview subjects were asked about how they and their team respond when adding dependencies between components the natural balancing act of coupling and code reuse was often brought up by the interview subjects, however how the teams reasoned about what was most important of these two differed. A few quotes have been gathered in figure 5.8, were different interview subjects explain how their team prioritize between code reuse and coupling. Some of the developers prioritize differently between code duplication and dependencies, leading to conclusion C5: *The developers are uncertain if they should prioritize code reuse or low coupling.*

From interview 8:
Interviewer: “What do you consider when you add dependencies between components?”
Subject 8: “We try to think about coupling and such, and to make sure that we are not too dependent on some component. However, when we need some functionality we always ask if there is some component out there that already provides this functionality. Because I am very much against code duplication, so I rather set up a dependency.”
From interview 11:
Interviewer: “Can you tell me about the process of developing new functionality and adding new components?”
Subject 11: “There is no general idea or process for the architecture, it's up to each team. However, our team tries to reduce the number of components, we have very many components right now.”
Interviewer: “When you reduce the number of components, does that result in bigger components with more dependencies or do you manage to keep those down as well?”
Subject 11: “Right now we are trying to reduce the number of dependencies as much as possible. But you always have to balance between having to duplicate code or having dependencies, but right now we are trying to remove as many dependencies as possible.”

Figure 5.8: Prioritization between code reuse and coupling

However, a more common response to the question was that the developers most often add dependencies when they feel like they need it, with no greater consideration to how that dependency would affect the overall architectural quality. A selection of quotes from the interviews related to this can be found in figure 5.9 that lead to the conclusion C6: *The architectural impact of dependencies is often not considered during development.*

From interview 2:
Interviewer: “What do you consider when adding dependencies between components or do you add dependencies whenever you need some other functionality.”
Subject 2: “Yes, that is usually what happens. If you need some functionality then you add a dependency you do not consider the architectural consequences of adding it”
From interview 4:
Subject 4: “The way I see it if there is a natural dependency in our domain then it's only natural to have it in our code too.”
From interview 6:
Subject 6: “Well, we want as a loosely coupled system as possible, but whenever we develop something that can be used in more than one place we place it within the business layer and we use it. But we have not thought to much about it.”
From interview 7:
Subject 7: “It's usually just, if we feel that we need the dependency we add it.”

Figure 5.9: Architectural impact of dependencies

During phase 5 of the interviews the specifics of components of interest that the interview subjects team have ownership of. Components of interests were components that in some way added larger amounts of complexity to the architecture or introduces other anomalies into the implemented architecture. During these conversations, duplicate components or components with overlapping functionality were often brought up. The reasons for this functionality duplication was varied, from phasing out of outdated components to problems with updating components that have incoming dependencies from outside the team.
In figure 5.10 a collections of interview quotes related to component duplication are gathered. Those quotes were the foundation for conclusion C7: *Dependencies between components contribute to difficulties when updating or removing these components. Often resulting in the duplication of components to facilitate migration, further increasing architectural complexity.*

From interview 4:
Interviewer: "'Discussing a component that have been duplicated' Why do you have two of the component active at the same time?"
Subject 4: "We have components that are used in multiple places, and if you realize that you need to add new functionality to that component and that by doing that the component will need such major refactoring that you will need to release a new major version of the component. And by doing this you will need to update maybe 15 other components that have dependencies on it, then it might be a bit too much work to update them all at once. Therefore you let the older version remain."
From interview 5:
Interviewer: "'Discussing two components that have overlapping functionality' Where these two added before you started working here or have they been added recently?"
Subject 5: "I have added the 'Newer component' recently, it provides new services. It is based on 'newer technology' while the 'older component' is based on 'outdated technologies', so we are moving towards the new and phasing out the older one."
From interview 6:
Interviewer: "I have noticed that you have many duplicates of the same components within the architecture, can you tell me why?"
Subject 6: "They are in transition."
Interviewer: "They are transitioning from one major version to another?"
Subject 6: "Exactly, and we have not had the time to update every component whenever we introduce breaking changes."

Figure 5.10: Component duplications

Another area of concern that was brought up when discussing dependencies between components was the cases were multiple teams share functionality through dependencies between components belonging to different teams. Quotes related to this have been gathered In figure 5.11 found on the next page. The developers explain that many of the components that exists in duplicate or are unnecessarily large have incoming dependencies from components owned by other teams. These quotes serves as the basis for conclusion C8: *Dependencies between components belonging to different teams introduce further difficulties and complexity to the architecture.* The quotes also further strengthen the before mentioned conclusion C7, that the dependencies existing between components makes the updating processing more difficult and encourages the duplication of components. Another aspect that was brought up during the investigation of particular components that add particular architectural violations or increased coupling was the adding of temporary components and the migration of ownership from one team to another.

The related quotes are gathered in figure 5.12 and covers the developers experience when receiving ownerships of components from other teams. The developers explain that it often leads to poorly maintained components and often break design rules. These quotes were the basis for conclusion C9: *Components with uncertain ownership add additional complexity and are not maintained.*

During the interview with subject 3 a few components that had been developed with little concern to the level of coupling and layer placement concerns were discussed. He/She then mentioned that they sometimes quickly developed component that had a short planned lifetime, were they have to cut corners to make a deadline. The quotes I refer to can be read in figure 5.13. Subject 3 also mentioned another integration component that were quickly developed to make use of a third party component that the team knew were going to be used for a limited time. These quotes were the basis for conclusion C10: *Components with a short planned lifetime are rushed and often have a longer lifetime than planned.*

From interview 11:
Interviewer: "'While discussing a particular component.' I have noticed that this component seems like a normal presentation component but it has a very large amount of outgoing dependencies. I was wondering if this was something your team had considered?"
Subject 11: "I had not thought about that, but if I were to work on anything that is related to that component I would write something new that provide that functionality. Something new that is more in line with our new way of thinking. I also wonder how much of that component we still make use of."
Interviewer: "Is it largely unused?"
Subject 11: "Yes, about 90% of its functionality. There is only a few leftover functions left that prevents us from removing it."
Interviewer: "So you could spend a sprint on removing it?"
Subject 11: "Yes, but the problem is that we are not the only team that uses its functionality."
Interviewer: "I was going to ask you about that too, does dependencies between teams negatively impact your work?"
Subject 11: "Absolutely, the shared components are a huge hassle. It is unclear who owns them and if we change them others gets impacted. So we try to avoid sharing components between teams."
From interview 2:
Interviewer: "Have you ever been prevented in making a change in a component or in some other way run into problems when working with a component due to the fact that other teams have dependencies on it?"
Subject 2: "Yes, I just remembered. 'component name' has two different versions active within the architecture."
Interviewer: "Why do you need two?"
Subject 2: "We introduced braking changes into the API that was also used by frontend developers in other teams. So, they used the old frontend component that still uses the not updated backend component. And since we could not expect everyone to update their frontend components when we introduced changes in the backend component we had to solve it by keeping two versions of the backend components up at the same time. We still have to have them both up, and we did the update last year."
From interview 7:
Subject 7: "'During a discussion of dependencies' We have wanted to work in our own context for a long time. Because whenever someone releases a major update to a component that we have dependencies on we don't get notified about that until just before release. If we worked within our own context we could delay that update until we were ready."

Figure 5.11: Component to component dependencies

From interview 4:
Interviewer: "I am curious about the 'component cluster name' components, they seem like a bunch of fairly tightly coupled components, can you tell me more about them?"
Subject 4: "I do not know much about them, they were developed by an outside team that was brought in. They were developed by a team of about two or three developers that only developed that cluster. So those components no not really belong to our team. And I believe that some of those developers had never worked for our company before, so that is why looks so different."
Subject 7 also mentioned that this cluster had been developed by an outside team and that they had not kept up with new design directives at the company, so when the outside team delivered the components they already where somewhat outdated and did not match the rest of the architecture. He/she also mentioned that they more or less have been left abandoned, but from time to time they discussed the task of updating them and putting them to use.

Figure 5.12: Receiving ownership

From interview 3:
Interviewer: "You mentioned the short lifetime of the component, is it often you develop new components that are planned to only exist for a limited time?"
Subject 3: "It is not very common, but we have a few components like that. We have for example 'Component name', it was a proof of concept component that was used during a temporary campaign, it was also developed under a tight time pressure."

Subject 3 also mentioned another integration component that where quickly developed to make use of a third party component that the team knew where going to be used for a limited time.

Interviewer: "Have you ever developed something that was going to exist for a short amount of time and then the component still remained longer than planned?"
Subject 3: "They always remain longer than planned, when we have decided to remove them we always let them stay a bit longer."

Figure 5.13: Component lifetime

5.5.3 Maintenance and Refactoring of Components

To understand to what degree and how developers maintain existing components, and if they consider the architectural impacts of components when maintaining and refactoring components a few questions regarding maintenance was asked during the phase 4 of the interview. A summary of the answers is outlined in table 5.2 as shown below, were the answers to the questions are categorized into three general attitudes towards the priority given towards performing maintenance and refactoring of components and the overall architecture. If the subject believed that they were able to prioritize and therefore take the time needed to perform maintenance and refactoring tasks the *Enough priority given* was checked. If the developer feels that refactoring and maintenance is only given enough priority when some functional or none-functional requirement are changed or added a check in the *Only when fulfilling new requirements* was checked. Finally, if the developer did not feel that maintenance and refactoring was given enough priority the *Not enough priority given* column was checked.

Subject number	Enough priority given	Only when fulfilling new requirements	Not enough priority given
1	✓		
2		✓	
3	✓		
4		✓	✓
5		✓	
6		✓	
7		✓	✓
8	✓		
9			✓
10		✓	
11	✓		
12	✓		
Total	5	4	3

Table 5.2: Priority given maintenance and refactoring tasks

A common experience reported by many of the developers interviewed were that they knew of problems within the components or within the overall architecture that were in the need of being addressed with maintenance work. But they could not address these issues unless they were given some other tasks related to the components in need of maintenance. Some of the quotes related to this issue is gathered in figure 5.14 that is shown on the next page. The issue of only getting an opportunity to update components when working on other new functionality was also brought up by interview subject 4. He/She also mentioned that this had in turn created additional problems since they have several components that they have to have several versions of running within the architecture at the same time. This is because they have not had the time to update all the components that have dependencies on the component they just made a none backwards compatible major version update to. This in turn added additional technical debt and additional complexity to the architecture. These quotes and observations were the basis for conclusion C11: *Maintenance work regarding known issues are often not prioritized unless they are combined with new requirements or fixing critical issues*, as well as conclusion C12: *Developers mostly perform maintenance and refactoring work on components when they have been assigned a none maintenance related task related to the components.* The quotes and observations also strengthen conclusions C7 and C8 both relating to the effects of outside team dependencies and component duplication.

From interview 2:
Interviewer: "Do your team assign enough of time on maintenance? Meaning, tasks related to updating existing components and refactoring."
Subject 2: "We have never assigned a task directly related to fix some issue in a component or structure. But they are rather related to some functionality, and when we are there and modifying code we notice that some part of the component need maintaining. It is fairly common that we see something and think, "Woops, we no longer do it like this" and then we fix it. But fixes like that are never a task themselves, since those kinds of problems often are not prioritized."
From interview 7
Interviewer: "'After talking about a known issue in need of maintenance' Do you think you get enough of time to work with maintenance issues like this?"
Subject 7: "Not as much time as we need."
Interviewer: "Do you have a lot of old technical debt like this that need to be addressed?"
Subject 7: "We have a lot of technical debt."
Interviewer: "Have it been like this for a long time?"
Subject 7: "Yes."
Interviewer: "When do you get opportunities to perform maintenance work?"
Subject 7: "There are two sides of this. We are fairly bad at bringing up issues in need of maintenance to our product owners, we usually just keep them in our heads or on some whiteboard somewhere. So, then our product owner does not know of these issues and can therefore not prioritizes them. So, then we only perform needed maintenance when we are working on some other task related to the components."

Figure 5.14: Maintenance

Many of the developers connected maintenance tasks to the task of writing automated tests for the functionality they had already implemented, an example of this is brought up by interview subject 10 is shown in figure 5.15.

From interview 10:
Interviewer: "Do you believe that the quality of your components or their services would improve if you got the opportunity to work more with maintenance?"
Subject 10: "I don't think so, I think that the quality would improve more if we got more time for other tasks?"
Interviewer: "What tasks are you thinking of?"
Subject 10: "Writing better automated tests."

Figure 5.15: Automated testing

Interview subject 9 also brought up that one of the primary factors for his/her team did not address known maintenance issues is that do not trust that their tests would detect if they were to introduce any faults when they updated these components. They also felt that they did not fully understand these components since they had been developed by developers that no longer worked within the company and their complexity prohibited them from fully understand them. When subject 9 was asked how his/her team reasoned about the task of understanding and updating these components he/she responded with that they mostly let the components be as long as they do not cause any problems. A similar sentiment was brought up by another developer, interview subject 10. When He/she was asked about why that developers team did not perform maintenance on components with known issues until the functionality of that component stops working.

From interview 10:
Interviewer: "How do you work with maintenance overall or do you only perform maintenance when you are working on some other functionality?"
Subject 10: "What usually happens is that we do not perform any maintenance tasks until a component stops working."

Figure 5.16: Maintenance of none functioning components

This quote is shown in figure 5.16. These observations served to conclude conclusion C13: *Inadequate tests and test data prevents developers from performing maintenance and refactorization of components.* Also, the quotes serve as further justification for conclusion C8, that is related to the dependencies between components belonging to different teams.

5.5.4 Layer Placement Related Violations

The architectural metrics CA and EV are measurements of different types of violations of the architectures layer rules. These violations are in essence different ways a components relationship to components outside or within the architecture that are not permitted in the components current layer placement. As seen in table 5.1 the number of these type of violations have increased in almost every event tracked by the static architecture software that was developed. When the components that contributes to these violations were investigated it was discovered that a majority of these components are what will be referred to in this thesis as *composite components*, illustrated in Figure 5.17. Composite components are components that are placed within the integration and presentation layer at the same time, and therefore can communicate with third-party components and support systems and expose functionality towards the frontend applications at the same time. This behavior is not allowed according to the architectures documentation. To find an understanding towards why newer components trend towards taking the form of composite components questions regarding the developer's attitudes towards the layer structure in the architecture as well as what is considered when exposing functionality towards the frontend application were asked during phase 3 of the interview.

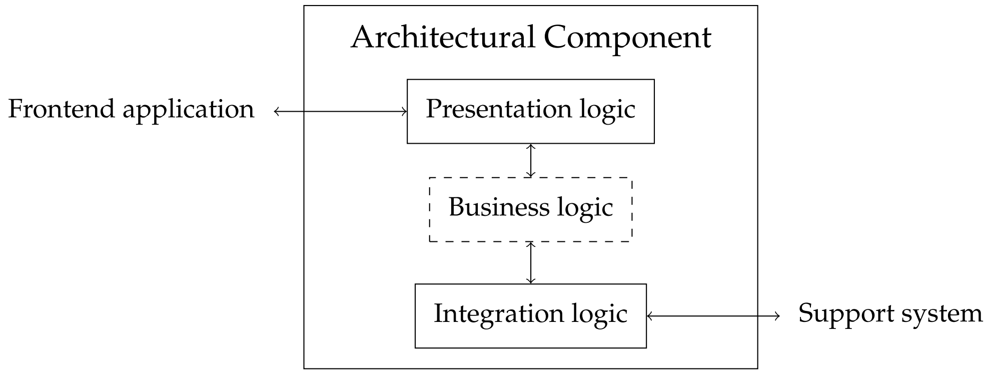

Figure 5.17: Composite Component

Subject 1, the frontend developer that was interviewed was aware of that backend components are placed within different layers, and the fact that some backend developers in his team often discuss the layer placement of different components. From the frontend developer's perspective only functionality exposed by the components are visible, and the layer placement and the relationships of the components providing this functionality is not considered when consuming the functionality.

During the interviews it was discovered that the most common approach to the layering of new business functionality was to merge it with an existing presentation or integration component instead of creating a new business logic component. A similar attitude was often taken with new presentation and integration functionality, but it was less common.
In figure 5.18 we can see an example of a developer, interview subject 2, that when developing new or maintaining components take no consideration to the components layer placement.

From interview 2:
Interviewer: "When you plan to add new components, what considerations are given to the layer placement of these components, or is only the functionality that they provide considered?"
Subject 2: "I do not know how they layer placement looks like in practice. Often we develop the business and presentation logic as one component and keep the integration functionality separate. But at least when I participated in the development of new components we usually don't consider layer placement very much."
Interviewer: "So when you develop new components no one is dictating that you'll need a presentation, business or integration component. You only consider the functionality that you need?"
Subject 2: "When I have participated we do not consider the layers that explicitly, but I cannot say that is never done."

Figure 5.18: No considerations for layer placements

This sentiment was shared by other more senior developers as well, as it is viewed by many of them as an outdated way of working within the architecture and the architectural platform. As was also indicated in the previous statements was that many of the developers do not consider the layer placement when planning and developing new components. This sentiment was shared by other more senior developers as well, as it is viewed by many of them as an outdated way of working within the architecture and platform. In figure 5.19 interview subject 11 and 12 explain that the concept of the three-layer structure is currently seen by the developers as outdated. Both of these senior developers explain that during the architectures early development the divide between the layers was heavily enforced by a physical divide between backend and frontend components. Interview subject 11 also mentioned that over time business layer has almost completely been merged with integration or presentation components by his/her team.

From interview 12:
Interviewer: "What is your opinion on the integration, presentation, business layer structure and the different roles the different layers have?"
Subject 12: "I can say that we do not really use that anymore. It was the plan in the beginning to divide up the architecture this way. Back then it was a physical separation in the platform, but due to some performance issues we decided to remove this separation. And now it's up to each time to decide how to organize their components, they have no longer any set requirements."
From interview 11:
Interviewer: "What is your thoughts regarding the presentation, business, integration structure?"
Subject 11: "In the beginning the though was that we would put all the presentation components in the frontend part of the architecture and have all backend components in the backend part of the architecture. But we later merge these two parts, and I think no one really thought too much about the structure of the components structure after that."

Figure 5.19: separation of layers

In figure 5.20 interview subject 12 expresses a similar sentiment were only components that aggregate functionality and data from multiple integration points are keept as business layer components. The others have been merged with either an appropriate presentation or integration component. These statement shows that a move away from the presentation, integration and business layer structure. Instead the developers have move towards the usage of composite components. The more junior developers are not too aware of the layer structure and the more senior developers see it as an outdated approach made for a legacy architecture. These observations were the basis for conclusion C14: *The layer structure has largely been abandoned by many of the development teams.*

Subject 12: "I want a downpipe like structure. Where a component is responsible for the logic all the way up, I do not approve of any unnatural mapping rules where you just transform data. But a businesses layer can be useful when aggregating data from multiple integration components."

Figure 5.20: Downpipe structure

To assess the developer's experiences and attitudes towards working with composite components within the architecture a few questions directed towards their development and maintenance. In figure 5.21 a few developer's response to questions related to the reasons given for developing composite components instead of either integration or presentation components. The reasons for abandoning the layer structure was varied, from making development faster and easier, to a perception of that the layer structure is outdated and over complicated. In this study the possible effects of this switch to composite components can have on future development within architecture is of special interest. Therefor a few follow-up questions about the further development of composite components was asked during the interviews.

From interview 6:
Interviewer: "Do you sometimes choose to merge presentation and integration logic into one bundle?"
subject 6: "It happens, but only when we are almost 100% certain that the component will not be further developed, then we sometimes merge them."
From interview 10:
Interviewer: "Do you often have to split up composite components?"
subject 10: "Yes, but it is very rare, and I believe we almost exclusively use composite components."
From interview 5:
Interviewer: "'The interviewer refers to a specific composite component' What did you consider when you decided to make this into one component? Was it just because you did not need any business logic in between?"
subject 5: "It seemed a bit excessive to make it multiple components, because it has no business logic to put between the integration and presentation logic. And it also seemed a bit excessive since it's a small component, but we also needed to finish the component as fast as possible."

Figure 5.21: Composite components

In figure 5.22 a few of the answers are shown. Some of the developers, like interview subject 5, mentioned that most composite components are very small, and they did not see any reason to split them up into presentation business or integration components.

From interview 5:
Interviewer: "'The interviewer refers to a specific composite component' Could you see a future need to split up this component, if its functionality where to increase."
Subject 5: "Maybe if the integration changes. The presentation part has no reason to increase in complexity. So no, I do not see a need to split it up, and the component is supposed to be as small as possible."
From interview 2:
Interviewer: "'The interviewer refers to a specific composite component that is very complicated.' Do you know why this component is like this?"
Subject 2: "So that we do not need to create multiple components."
Interviewer: "Has the component always been this complex, or has its complexity grown over time?"
Subject 2: "It has grown in complexity, but it has always been a composite component. I do not know if it used to use other components through OSGi, but it has such dependencies now."

Figure 5.22: Breaking up composite components

The component referred to in the quotes gathered above was also discussed in the interview with subject 3, who have been working with this component during its entire lifetime. He/She explained that the component had started as a simple composite component that moved data between the frontend application and a third-party component, however it have become more complex over time since another team wanted to integrate towards the same third-party component. The component then needed to provide its functionality through OSGi dependencies to other components with additional business logic to manage the need of other teams. Subject 3 also mentioned that he/she saw a need to split this component into the three-layer structure, but it had simply not been prioritized.
In figure 5.23 responses form interview subject 6 and 7 explain that they follow the older presentation-business-integration when developing new functionality in the cases were they do not find an appropriate already existing component to place the functionality in.

From interview 6:
Interviewer: "When you develop new components, do you consider the layer placement of that component"
Subject 6: "I am fairly new here, but from what I have learned that you place component that integrate towards subsystems in the integration layer, and those components should basically only get data. If we need to operate on that data we do it in the business layer and then we move that data to the presentation layer that serves logical separation from the frontend application. Also, if we need to do any security or validation we do it in the presentation layer."
From interview 7:
Interviewer: "Can you tell me about the process of adding new functionality, new bundles and such, into the architecture?"
Subject 7: "That depends a bit on what kind of functionality that will be added, if it is a new integration we add a new component for that. Then we add one or two layers above that depending on if we any business logic. That is usually how we do it, otherwise we ask ourselves if we can put the new functionality in some already existing component."

Figure 5.23: Layer placement considerations

5.5.5 The Increase of Special Component Access Violations

As seen in table 5.1 the number of special component violations recorded in the architecture have increased with almost every new release of the implemented architecture. To understand the reasoning behind not using the dedicated interface components when interfacing with outside components and internal support systems questions regarding the usage of these interfaces were included in architectural considerations phase of the interview. As can be read in figure 5.24 The frontend and fullstack developers had no knowledge about the interface components.

From interview 1:
Interviewer: "If I say 'Interface component', what does that say to you?"
Subject 1: "Not a lot, I read about it in the 'Company chat' from time to time."
Interviewer: "But it is nothing that is used or seen from the frontend application?"
Subject 1: "No."
From interview 5:
Interviewer: "Do you make use of the 'Interface component'?"
Subject 2: "I do not believe we do, I have heard it mentioned on backend meetings but I do not really know what it is, do you know?"

Figure 5.24: Interface component

When subject 2 was asked why he/she believed that his/her team did not make use of the interface components, he/she believed that the components that the team were responsible for did not need them and that the information regarding the usages of the interface components had not reached the developers in the team. Even among developers that have experience implementing the interface components, like subject 10, 11 and 12, believe that the lack of knowledge of the existence or the usage of the interface component is the primary cause for the increased number of special component access violations.
Another common theme that was brought up by the interview subjects during the interviews were the fact that the many developers did not feel like they knew how they would properly configure the interface components if they were to implement it. While the interface component provides a default configuration the experience of many of the interview subjects, subjects 3, 4, 8 and 9, were that an improperly configured interface component will cause more harm than good. In figure 5.25 interview subject 8 explains that he/she believed that the difficulty to properly configure the interface component was the primary reason for it often not being used. These observations and quotes served as the basis for conclusion C15: *The primary reasons for increase in special component access violations is the lack of knowledge of the interface functionality provided by the architectural platform.* The quotes also served as the basis for conclusion C16: *Difficulties with properly configuring the interface functionality is a contributing factor to why developers choose not to use it.*

From interview 8:
Interviewer: "When you write integration components that interfaces towards support systems or third-party systems what considerations are taken towards the usage of 'Interface component'"
Subject 8: "We do not think about that a lot, at least not in our team. It is very hard to know how to configure the 'interface component'. How much traffic can the support system take? Because sometimes I see discussions in the 'Company chat' when they have had issues with some support system it is often because the 'Interface component' have stopped the usage of the support system due to the upper limit in the default configuration. And the most common solution to this is just to set a higher upper limit."
Subject 8: "So the primary reason for why we don't use the 'Interface component' is because we do not know what to configure the 'Interface component' towards the support systems we use."

Figure 5.25: Interface configuration

Another aspect that was revealed during the interviews was the fact that the usage of the interface component is no longer part of the requirements given to developers when they are tasked with developing new functionality. In figure 5.26 subject 3 and 4 explain that they do not receive and requirement from stakeholders to make use of the interface component anymore.

From interview 4:
Interviewer: "How is your teams attitude towards the 'Interface component'?"
Subject 4: "I do not even know to what degree it is used anymore. In the beginning, we were required to use it everywhere. But at the same time no one knew how to configure it back then."
From interview 3:
Interviewer: "Do you feel like you and your team do not know how to make use of the 'Interface component' or why you should use it or do you feel like your team is not required to do so?"
Subject 3: "We know how to use it and what it is used for but we do not receive any requirements from our stakeholders or product owners to use it. Instead it is something we have to consider when we develop, it is on us. But it would be nice to get some more architectural point of view when we develop, and some references, maybe reference implementation to refer too."

Figure 5.26: Interface requirements

A final observation is that the rapid influx of new developers, increase of team sizes and the existing of more senior developers may affect the usage of the interface components, since it is a more obscure part of the company's architecture. In figure 5.27 interview subject 4 and 11 explain that newer developers are often not instructed in the use of the interface component. And as the size of the development teams and the number of new teams increase it is harder for the more senior developers to get an architectural overview to notice such violations. These quotes serve as the basis for conclusion C3: *The number of new developers have increased and few senior developers have remained since the introduction of the architecture.*
In Table 5.3 a summary of the reasons given by the different interview subjects for why the

From interview 11:
Interviewer: "It seems to me like the knowledge of the 'Interface component' is very varied, why do you think that is?"
Subject 11: "That is probably because there is so many developers now, and there is always new developers arriving. I think we often underestimate how much developers need to learn to develop here, and especially now when we often expect developers to both work on the backend and frontend and I think the knowledge of the architecture gets thinner then."
From interview 4:
Subject 4: "We used to be fewer back then and we had smaller teams. So back then I had an overview of every component my team where responsible for since I have been here so long and where a senior developer. But now we are more spread out and some developer sitting in the corner over there that can do something that I do not know anything about."

Figure 5.27: Interface knowledge

number of special component violations have gone up in later releases of the implemented architecture. A check in the *Proper usage* columns indicate that they believe that their team have used the interface components properly. A check in the *Lack of knowledge* column means that the subject believes that the developers, either that their own knowledge or their team or other teams knowledge of the usage or existence of the interface components. A check in the *Insufficient documentation* indicate that the developer believe that the interface components is not properly documented and that contribute to the fact that it is not used. If the developer believes that either that the fact that developers do not know how to configure the interface component or that an improperly configured interface component causes more harm than no interface component is indicated with a check in the *Configuration issues* column. A check in the *No requirements* column indicates that the developer believes that the lack of requirements from stakeholders are the reason the interface components are not used, and finally, if the *Oversight* column is check indicates that the developer believe that it's simply an oversight or mistake that the interface components are not used.

Subject number	Proper usage	Lack of knowledge	Insufficient documentation	Configuration issues	No requirements	Oversight
1		✓				
2		✓				
3			✓	✓	✓	
4				✓	✓	
5		✓			✓	
6		✓				
7		✓	✓			✓
8				✓		
9		✓	✓	✓		
10	✓					✓
11	✓	✓				
12		✓				✓
Total	2	8	3	4	3	3

Table 5.3: Reasons given for the increase of special component violations

5.5.6 Development Methodology and the Release Cycle

To better understand the conditions that the developers work under when working within the architecture and what effects those conditions might have on architectural quality. To this end, a few questions were asked in the later part of phase 4 of the interview regarding the subject's perspective on how the processes of releasing components as well as how the subject verify the quality of components and their impact on the overall architecture. The company currently employ a release cycle were new code is released at scheduled times. These scheduled release dates are detached from each team's development cycle as the team has the autonomy to structure their own development priorities. The questions were focused on how they felt that the current release cycle effected the quality of their code, if they often felt they had to release hotfixes for *rushed components* and how they felt about the state of the tests they ran before releasing the components. The interview subject's perspective on how this release cycle effects the quality of their components and the impact on the overall architecture is outlined in table 5.4. While all the interviewed subjects said that they believed that the ability for them to release whenever their components were ready they had different views on what impact the current release cycle have on the quality of their components. If the interview subject believed that the release cycle does not have any major negative effects on the quality of their components or the architecture the *No major adverse effects* column was checked. If the developer believed that the release cycle results in them having to rush the development of components and that it negatively affects their quality the Rushed components column was checked. Finally, if the interview subject believed that the release cycle results in the developers having to release components later then they need to be the *Delayed releases* column was checked.

Subject number	No major adverse effects	Rushed components	Delayed releases
1		✓	
2			✓
3		✓	
4		✓	✓
5			✓
6		✓	
7		✓	
8	✓		
9		✓	
10		✓	
11	✓		
12		✓	
Total	2	8	3

Table 5.4: Effects of the release cycle

A recurring observation gathered from the interviews subjects were that they preserved a negative discrepancy between the release cycle and the team's own development cycle. In figure 5.28 subject 1 and 4 explain that the current release cycle makes it much harder to release new functionality. Primarily that they often have to unnecessarily delay components since they are not ready for a particular release date.

Another recurring observation was that the rushing of components leads to the need to

From interview 1:
Interviewer: "'After discussing the release process' So it's the newest components that sometimes gets rushed?"
Subject 1: "Yes, but that depends on what is prioritized by the product owner. But I believe that the biggest problem is the release cycles slow process, it causes fairly many problems. If you end up in the wrong phase then you have very little time until the next release. And that is just bad luck, if we would have started working on that component a week earlier then we would have had plenty of time."
From interview 4:
Interviewer: "What is your experience of the release cycle?"
Subject 4: "It does not help, it's terrible. The release cycle is from the waterfall method but here we are trying to work after the agile method. We often end up missing the release train and then we have to wait until the next one and that causes a bunch of problems. We would like to be able to release when the functionality is done."

Figure 5.28: Release cycle

immediately performing maintenance tasks on these components, since they were released before they were ready. In figure 5.29 interview subject 7 and 10 explain that new functionality often has to be unnecessarily rushed to meet a release deadline due to the fact that the teams sprint is out of sync with the release cycle. During the rushed development the developers sometimes have to use shortcuts to make the deadline, these shortcuts are also often not rectified in later patches or hotfixes until they cause a major problem.
The quotes shown in figures 5.28 and 5.29 serve as the basis for conclusion C17: *The release cycle contribute to the rushed development of components.*

From interview 7:
Interviewer: "Do you feel that the limitation in the release cycle is that you have to rush components or that the release of already finished component have to be delayed?"
Subject 7: "The rushing for a release is a bigger problem, because otherwise you would have to wait several weeks for the next one."
Interviewer: "When you rush components, do you have to patch or maintain them afterwards and if so, do you get the opportunity to do so or do you have to move on to the next functional requirement?"
Subject 7: "Usually we have to move on"
Interviewer: "What kind of improvements are needed in the components that have been rushed?"
Subject 7: "Mostly to remove shortcuts that have been made to rush them out, and change some stuff that make them easier to manage and maintain. Because that will lead to problems in the future."
From interview 10:
Interviewer: "Do you feel that you need to rush some components for a certain release and then only have to release a hotfix for that component later?"
Subject 10: "Yes, that happens because it is hard to work with a release cycle that is completely removed from the sprint. You almost always end up out of sync. When we priories some new functionality we often realize that we only have one week to work on this functionality, so we should really have started work on this functionality during the last sprint."

Figure 5.29: Release cycle rushing

Another topic that the interview subjects often brought up when discussing the release cycle and the process of applying patches and hotfixes was the developer's dissatisfaction with the current state of the automated testing. As mentioned in the previous sections some of the developers believed that a more proper test suite and testing environment would have a greater positive effect then more time and priority made for maintenance tasks. A similar observation was made by some developers when discussing the reasons for having to apply post-release hotfixes, relating to the testing suite and in particular the data it uses.
In figure 5.30 an excerpt from the interview with subject 8 can be read were subject 8 mentioned that the primary reason for the need to patch components short after release was the lack of proper test data for automated testing of the new components. Subject 4 also expressed that one of the components that have required continuous patching is not only that it is a component with tight time to market requirements and high domain complexity, but it also has no testing data that represent what it is expected to handle when the component is put into production. Subject 6 also mentioned that they often cannot properly test some integration components since they can not represent data from third party components within their testing environment and therefore first get an opportunity to test within production. These observations served as the basis for conclusion C18: *Rushed components often have to immediately undergo maintenance after release.*

From interview 8:
Interviewer: "Have you needed to release many hotfixes?"
Subject 8: "It have been a couple, I guess it depends on how you define many, but there have been a few."
Interviewer: "Do you have any insight on why they have been needed?"
Subject 8: "I would say that most commonly it's because of the data used during testing did not match the data in production. It would be terrific if we had production like data that we could test with, and preferably production like data that did not change to often. Our test data is a huge problem."

Figure 5.30: Release cycle hotfixes

5.6 Microservice Reengineering

This section serves to outline the results of the microservice reengineering process. The migration was done manually in three stages were different rules were used to determine what OSGi components were to be merged and transformed into microservice component. The changing rules are listed in section 4.7.3.

5.6.1 Static Architectural Analysis of Migrated architecture

In this section, the results of the static architectural analysis of the planned migrated architecture will be shown. The results of the migration are shown in tables 5.5 and 5.6 below. The tables below contain some additional columns as compared to table 5.1, outlining the differences of quality metrics in the events of major change. The column Δ *MS-c* outlines the number of microservice components that have been introduced during the migration. The Δ *Total-C* column shows the difference in the total number of components present in the architecture, including both OSGi and microservice components. The column Δ *WD* shows the number of dependencies from a new microservice component to a OSGi component through a webservice. The final new metric is OSGi-U, this metric measures the number of OSGi components that would need to be updated to provide their OSGi-services through webservices. The results presented in the tables are written as the difference in quality metrics recorded in the latest version of the implemented architecture as shown in the row for version 56 in the table A2T1 found in appendix A.2. A table showing the measured metrics of all the migration steps are shown in the table A2T2 found in appendix A.2.

Migration	Δ OSGi-C	Δ MS-C	Δ **Total-C**	Δ CA	Δ CD	Δ EV	Δ SCA	Δ **SA**
1	-91	20	-71	-27	0	-8	-13	-21
2	-155	35	-120	-33	-2	-10	-18	-30
3	-231	98	-133	0	-2	-18	-18	-45

Table 5.5: Architectural quality metrics difference (OSGi-C to SA)

Migration	Δ DD	Δ MD	Δ WD	Δ ID	Δ DOC	Δ SD	Δ IDD	Δ IID	Δ **CoA**	Δ OSGi-U
1	-359	0	173	-657	153	15	-160	-650	-1653	72
2	-570	42	238	-949	153	15	-248	-938	-2425	76
3	-691	345	0	-1025	153	15	-257	-1014	-2642	76

Table 5.6: Architectural quality metrics difference (DD to OSGi-U)

6 Discussion

In this chapter the results documented in the results chapter will be discussed. First, the results from the static architectural analysis, interviews and microservice reengineering will be discussed. As well as how well the conclusions from the interviews match the results from the analysis and how these results stack up to what has been written in other studies, papers and literature. Then a critical analysis of the methods used during this thesis as well as any threats to the validity of the results. Finally, this chapter will end with a discussion of the results in a wider context as well as what future work can be conducted to expand the results of this study and topic.

6.1 Architectural Analysis

This section will serve to summarize the conclusions listed in section 5.5 in the results chapter together with the results of the static architectural analysis as outlined in section 5.3. The conclusions and erosion metrics will also be related to relevant theory, other conclusions, and architectural context.

6.1.1 The Abandonment of Architectural Design Decisions

The fact that some parts of the stated architectural design has been abandoned is made clear by the consistent increase of consistency violations as seen in figure 5.2. As well as the consistent increase of structural violations seen in figure 5.3 and interview conclusion C14, the layer structure has been abandoned by many of the developers, as well as conclusions C15 and C16, that the platform interface component is largely unused due to the ignorance of its existence and how to make use of it. This abandonment of the three-layer structure is seemingly largely intentional. When architects and stakeholders stopped requiring components to follow the three-layer structure, and when the architectural platform was updated to no longer require developers to separate presentation and integration components as separate bundles the teams were free to decide how to structure their components. Most but not all teams then choose to evolve in a greater or lesser extent their architectures towards the usage of composite components instead. Within the scope of this thesis work it is hard to say that the evolution towards composite components is necessarily a negative development but the evolution has had a few effects. The aforementioned increase of the number of structural violations, in particular exposure violations, and consistency violations is the first effect of this evolution. These violations primarily contribute towards a general ambiguity of a components functions, since it has no obvious role within the architecture and what its function it has. Another effect is the potential increased risk of accumulation of complexity and coupling of certain components. These erosion causes and their effects are consistent with the observations made by *Van Gurp* and *Jan Boschs* in their case studies [48, 49], as well as the paper *Does code decay?* [11]. Mainly the erosion causes of the *vaporized design of design decisions* [48] and the *lack of attention to design during evolution* [11, 49] leads to further *Uncertainty about specifications*, something that was also brought up during interviews and resulted in the interview conclusion C4. Where the abandonment or lack of knowledge of earlier design decisions during evolution leads to further uncertainty for new developers that study the code and make design decision during future development. Another observed consequence of the well as an increase of software *kludges* [11], were developers knowingly introduce inelegant, messy or inefficient changes into the code. In particular as it relates to large, complex and highly coupled composite components.

A more concerning consequence of the abandonment of the design is the aforementioned increase of *Special component access violations*, as this contributes to additional security and stability risks. As mentioned, conclusions C15 and C16 from the interviews point towards a lack of knowledge of the interface functionality and how to properly implement it, as well as a level of mistrust from the developer's towards the interface component. There appears to be a discrepancy between many of the developers knowledge and views of the interface components and the architects view on it. As the interviews also indicated that the documentation of the interface component is lacking combined with conclusions C3, that there have been a recent larger turnover of developers and increase in team sizes, C1 that the architectural documentation is primarily used during the developer's familiarization phase and C2, that architectural knowledge is primarily acquired through close colleagues the reasons for the increase of special access violations in newer components become clearer. Both inadequate documentation and the influx of developers that are inexperienced with working within the architecture are well cited causes of software erosion [11, 35, 49].

6.1.2 Coupling Considerations During Development

One of the quality attributes that the original architectural design was supposed to achieve was a loosely coupled and modular design, and since this is one of the primary quality attributes of a service-oriented architecture [47] a degradation of this quality during evolution was naturally considered architectural erosion. As seen in figure 5.4 and table A2T1 found in appendix A.2 the level of coupling has been increasing as the architecture have been updated. It is also important to note that the level of coupling measured was not directly connected to the number of components present in the architecture, meaning that even when the architecture was evolved, and the number of components stayed the same the number of dependencies between then still increased. The fact that the level of coupling has increased and the level of modularity as new functionality has been added over time is not surprising, it is a well-documented effect of erosion [11] and have been documented as a consequence of the erosion of object-oriented systems [22]. While many papers [38, 53, 25, 52] proposes methods and metrics for evaluating and measuring coupling in a service-oriented architecture they all have still not been properly tested when evaluating the quality of a real implemented service-oriented architecture. There are not many papers evaluating the effects of evolution and erosion on coupling and modularity of a implemented service-oriented architectures at least one company have been reported to have developed a architecture with a much higher number of dependencies after a migration to SOA as compared to their legacy system [42] while the rest investigated in the same paper seemingly have managed to create a more loosely coupled architecture in there. One of the contributing factors to the increase of coupling is the

encouragement of code reuse through dependencies. The documentation at the company and by theoretical SOA design [34] the sharing of functionality through the consumption of services, however due to the nature of the fact that OSGi dependencies have to be resolved for a component to function the level of coupling increased through dependencies might be worse than if it were implemented through say web services.
As observed during the developer interviews, conclusions C6 and C5, that developers often do not consider coupling during development and that they are often uncertain if they should prioritize low coupling or few occurrences of code reuse. These conclusions seem to match and explain some of the increasing levels of coupling observed during the static architectural analysis. The effects of this increasing level of coupling were also brought up during the interviews, with conclusion C7 that the number of dependencies to certain components have introduced difficulties when updating and removing components and the problems get even worse when the dependencies are between components belonging to different teams, as concluded by conclusion C8. These conclusions seemingly confirm the consequences that one would expect when the coupling of an implemented architecture.

6.1.3 Nonoptimal Development Conditions

Not all contributing factors towards architectural erosion is related to its design. The conditions present during its development, evolution and maintenance, some of the non-design related contributing factors observed is covered in this section.

The Release Cycle

The currently release cycle, allowing new code to be release into the production environment at certain predetermined times, has as concluded by C17, that the release cycle contributes to the rushing of components, have introduced additional time pressures into the development of components. Time pressures during development is a known cause of erosion [35, 49] as it encourages quick and dirty solutions and does not give enough time for reviews of the code before it is released. A consequence of this rushing was brought up by conclusion C18, that rushed component sometimes had to be maintained directly after release. It is also important to note that many of the developers felt that the current release cycle introduced arbitrary time pressures were the time pressure is to keep up with the release cycle and not to complete their functionality on time.

Maintenance

The developers interviewed were often aware of components that were in need of maintenance or refactoring. Often larger maintenance tasks were created by changes to components with large amounts of incoming dependencies that would require the dependent components to be updated as well. In some cases, even small changes would result in a large number of components in the need of maintenance. When this happened, some developers did choose to employ quick fixes, like for instance letting multiple versions of the same component co-exist within the architecture, in order to not having to perform the maintenance at that time. Quick fixes and other unresolved maintenance issues is one of the common causes of architectural erosion [48, 49] and while these quick fixes often by themselves do not introduce many issues into the architecture they can over time grow. While components in need of maintenance were often known, the maintenance tasks to rectify them were, as concluded by conclusion C11, in some teams not prioritized unless tied to the fulfilment of another new functional requirement or the fixing of a critical bug or other critical issue. However, as concluded by C12, maintenance work is often conducted informally when a developer is working on some other task related to the components in need of maintenance. While it is a good practice to fix adjacent issues when working on a component this alone cannot be the maintenance strategy.

Testing and Testing data

As noted in conclusion C13, that inadequate tests and test data prevents developers from performing maintenance tasks, and in other interviews that an increased focus on developing a better test suite would greatly improve the ability of the developers to better work within the architecture and to comply with its design. A well develop suite of unit tests, regression tests and architecture compliance tests is recommended in literature to prevent architectural erosion [50].

6.2 Recommendations for Prevention of Architectural Erosion

As the company is planning a migration from a Service-oriented architecture to a new Microservice architecture the company is interested in taking steps to prevent or at least mitigate the kinds of architectural erosion observed in the future. To this end the following recommendations have been compiled:

- **Regression testing**
  As concluded in conclusion C13 as well as furthered expanded upon in section 6.1.3, the current state of the testing at the company has negative impacts on both the software quality of the components and the quality of the overall architecture as it prevents developers from performing needed maintenance and refactorization with the confidence knowing that the refactorization will not negatively impact any functionality provided by the components. As a consequence of this uncertainty the developers sometimes choose not to perform the maintenance until absolutely needed. To give the developers the confidence to maintain and refactor components early and often, so that erosion factors can be reversed before they grow a full suite of regression testing suite is recommended. Automated testing is a commonly recommended in theory [19, 49] for the same reasons given here. As the company is planning to migrate all legacy components to the new architecture, components that currently often have insufficient of tests written for them, there is a opportunity to include test requirements as a part of the migration requirements.

- **Complete and proper migration**
  While it is unrealistic to expect that the entire current implemented architecture can be migrated quickly. Regardless, the company should prioritize the migration process highly since as long as any of the components in the current architecture remain unmigrated the entire platform must still be maintained as well as any weaknesses present in its design or implementation will remain.

- **Architectural compliance checks**
  In order to combat important architectural design violations, such as the special access component violations, architectural compliance checks can be included in the test suites or be done manually by architects before a release. Manual architectural compliance methodology, like the different reflexion model techniques [27, 30] have been used to map discrepancies between intended and implemented architecture within the industry as well as automatic alternatives like the modelling tool developed for this thesis, see section 4.2, and others presented in theory [9, 54]. While wide spread adoption of automatic compliance checking software has not been wide within the industry [9], mostly due to the high level of effort to construct and maintain but its efficacy for detection of early instances of erosion so the cost of either rectify the detected defect or to purposefully evolve the architectural design to allow it is low.

- **Dependency monitoring**
  The planned microservice architectural design prioritizes as independent components

as possible, a microservice to microservice dependency should only be added when absolutely necessary. Therefore it is important to keep track of all active dependencies between microservices as well as microservice dependencies on yet to be migrated legacy components. A automated system to gather and visualize dependencies between components can help to slow down the increasing rate of coupling complexity, see figure 5.4, as well as begin to the address the issues brought up by interview conclusion C6, that the effects of a dependency is not considered before introduced. As well as give the developers more information regarding the level of coupling of certain components to mitigate the difficulties brought up by interviews conclusion C7, that dependencies introduce difficulties when modifying dependent components and conclusion C8, that dependencies between teams introduce even more difficulties for developers. A good overview of each components dependencies will also help to keep down maintenance costs and help divert dependency related issues [1].

- **Improved release cycle**
  As mentioned in section 6.1.3 the currently employed release cycle at the company lead to the rushed development of components. To address this the company should take steps to either remove the release cycle or to increase the rate of available releases to allow development teams to release whenever they feel ready to release or at least that a missed release will not substantially delay new functionality.

- **Clear and well documented architectural design**
  As indicted by interview conclusion C4, some of the developers are uncertain about intended architectural design during development. Couple this with the interview conclusions C15 and C16, where an important architectural security component often went unused due to the lack of knowledge of its existence and how to configure it. With this in mind it is easy to conclude that the architects architectural design where not documented or communicated well enough. Imprecise requirements [11], both functional and architectural design requirements, and more importantly the developers lack of knowledge of intended design [11, 49] contribute to the erosion of the architecture and its quality during its evolution. To mitigate this, it is important not only to document intended architectural design, but also keep it up to date as the architectural design is updated during evolution, if a design decision is not documented it is not done [35]. Keeping up to date architectural documentation is one of the most widely used method of architectural erosion minimization, and while it alone cannot significantly prevent erosion it is fairly cost effective [9]. In order to prevent the problem concluded by C1, that the developer only make use of the documentation during the initial set up and learning phases only. It is recommended to make sure that it is up-to-date with implemented architecture, focusing on providing a big picture overview and convey design goals, designed to serve the developers needs with examples were appropriate [35, 39].

6.3 Recommendations for Repair of Architectural Erosion

As the current architecture is planned to be replace it is not as important to reverse the erosion present in the current architecture. However, had the company not decided to invest in a new backend architecture they could employ repair methods to reverse the current levels of erosion. It is therefore important to examine the methods that are available to reverse architectural erosion presented in literature [50, 9]:

- **Redevelopment**
  While this recommendation is slightly redundant as the migration process has already been started. However, the redevelopment of already existing and eroded components is often the most effective way to reverse architectural erosion once it has been detected [49]. This process is fairly work intensive and sometimes hard to justify to

stakeholders as it does not introduce any new functionality and only serve to improve architectural quality.

- **Prioritize maintenance**
As mentioned in section 6.1.3, the developers often know of components that are in need of maintenance but they do not feel like they have the time to properly address. The quick fixes and workarounds the developers have introduced into the architecture to deal with this can often be seen as architectural erosion as they often take the form of duplicated components, unnecessary coupling or bloated components. By prioritizing the maintenance work continuously these smaller infractions can be rectified before they grow into larger and more entrenched architectural problems.

- **Prioritize testing of legacy components**
By implementing adequate testing suites, environments and data for all legacy components would greatly increase the developer's ability to perform needed maintenance and refactoring work.

- **Update and refine architectural design**
As stated in section 6.1.1, much of the original design has over time been abandoned. Some of the changes, the move towards composite components for instance, could be seen as an architectural evolution as the design has been changed in a way to make it easier for developers to develop new functionality and require the developers to develop and maintain fewer components to provide the same functionality. Composite components have not been documented in the architectural documentation and no real set of architectural design rules have been defined for their usage. To what extent is it appropriate for a composite component to provide functionality to the fronted application? To what extent can a composite component provide functionality to other components through OSGi? Can multiple composite components integrate towards the same support system? How much functionality is it appropriate for one composite component to provide, when is it more appropriate to split it up into multiple components. All of these design decisions have to be made and they have to be documented. A strong focus on design from the architects can be a way to begin reversing architectural erosion [49] as it gives developers an up-to-date architectural design to work with.

- **Acquire architecture overview**
As recommended in the prevention section 6.2 the usage of compliance checking, dependency monitoring and architectural visualization tools could give developers and architects needed architectural overview. This overview is not only useful for developers and architects to make better design decision but can also serve as an early warning system for architectural problems before they are fully implemented [50].

- **Reduce the architectural knowledge gap**
In order to reverse the observed architectural erosion, the knowledge gap has to be reduced. In particular the existence of and the usage of the special interface component has to be spread among the developers in order to begin the work of rectifying the increased level of special access component violations.

6.4 The Results of the Microservice Reengineering

If one were just to look at the results of the microservice reengineering as shown in table 5.5 and 5.6 as well as table A2T2 found in appendix A.2, the microservice reengineering could be seen as a complete success. Since all complexity, coupling and structural problem metrics have been reduced, which is the goal of architectural redevelopment [50] and the microservice architectural style in particular [31]. However, due to the limitations of the microservice reengineering methodology used in this thesis the results should be seen as highly hypothetical, especially those recorded from migration step 2 and 3, and should be seen as a starting off point for further reengineering work. While the process of identifying dependencies clusters to migrate as microservice is recommended within the literature [31, 12] it is recommended as a part of the toolbox that architects and other experts uses when they can make design decisions, and not by itself. Some aspects that could have improved the results and how well they would accentually represent a migration of the current implemented architecture.

- **Lack of component duplication**
  During migration step 2 and 3 the restriction on microservice to microservice dependencies were lifted. This introduces a problem when migrating OSGi components that have multiple migrated microservice components dependencies on it, as it will require a design decision to be made, should the OSGi component be migrated as its own microservice component or be duplicated as a component within all the dependent microservices? While a duplication could result in a less coupled architecture but might not be appropriate in every instance, for example when an integration to some support system is used.

- **Lack of component knowledge**
  Since the person performing the cluster transformation only had a limited knowledge of the components and their design and function, clustering decisions could only be made based on the components stated dependencies. In reality, other criteria could be used as well to determine what components belong together.

- **Lack of future platform knowledge**
  Much as the current architectural platform provides some functionality to the components running with in it, the planned future microservice platform does as well. There exist plans at the company to integrate the functionality that is provided by some of the more highly coupled components directly into the platform. However, this could not be taken into account when performing the microservice reengineering for this study.

6.5 Recommendations for Microservice Design

This section will serve to give some recommendations for what the company should keep in mind as they design their migrated microservice components from an architectural perspective.

- **High cohesion, Low coupling**
  As mentioned in the section 2.8 a microservice component should be loosely coupled with the rest of the architecture. This is not only a core tenet in microservice architectural design but also important for providing the modular architectural design the company is looking for. To achieve this the developers should carefully consider how functionality should be distributed between components as well as how to share the functionally between them.

- **Break away from the shared resources**
  One of the carry overs from the monolithic architectural design is the shared resources, such as databases, that is used by all parts of the architecture in a shared instance [31]. This is a large source of coupling and the resource should if possible instead be run as independent instances in each microservice component that needs them. In the example of databases each component should not only run its own instance but also have its own set of tables that are not synchronized between components unless it is absolutely necessary.

- **Make code sharing a design decision**
  When possible, a microservice component should have no dependencies to other microservice components. This is in reality not possible since the costs of redeveloping the same functionality is not always justifiable. There are however multiple ways to share code between components and one method might not always be appropriate for all situations. Therefore, how functionally is shared between components should be considered an architectural design decision. Following is a list of three code sharing methods and some considerations when they should be chosen:

  – **Component to component dependency**
    A microservice can expose its functionality through some web interface [51], such as REST for example, they can provide functionality to other components. Unlike OSGi dependencies however these dependencies are run-time dependencies that, depending on implementation, does not need to be satisfied for the component to function. These looser style of coupling comes with its own set of difficulties as unless kept track of they can become a source for regression problems and architectural confusion [31, 50].

  – **Binary dependencies**
    A binary dependency is when a component make use of a pre-compiled binary file. This file can provide some functionality that needs to be shared between components, thus a team can distribute this file to any component that need that functionality. This approach is a fairly quick and easy way to share code between components, however it does not allow for the component to modify the source code within it to that components need. It is also up to the developers to make sure that they keep their binaries up to date and implement whatever changes these updates require. Binary dependencies also allow different components to run different versions of the same functionality, this should be done with care however, as any update regarding security should be implemented immediately.

  – **Clone code**
    As the teams already uses a version control system to manage their code repositories for their components they can allow different team-members and other teams to clone, fork, pull and push updates from these repositories in order to share code and functionality. The forking of a repository allows teams to work simultaneously and in parallel on the same shared piece of code, in this way each component can get a customized version of the code.

- **Record dependencies**
  Due to the nature of the loser coupling of dependencies through web interfaces it might be harder to track [31] as opposed as the well-defined and easily readable OSGi dependencies. This challenge makes it more important to make sure that when a microservice to microservice dependency is set up it needs to be well defined and documented.

6.6 The Work in a Wider Context

The purpose of this section is to discuss, in brief, this study and its results in a wider context. In particular the ethical and societal aspects of the thesis and it's results.

6.6.1 The Data Collected

The conclusions drawn from the interviews and the data collected from the static architectural analysis can be used by developers and primarily architects to select areas of particular quality erosion. The architects have expressed particular interest in rectifying any special access violations in order to ensure that they get rectified as soon as possible. This to ensure that any vulnerabilities that could result in harm done to the application and its up-time gets dealt with. The data collected and the conclusions drawn will hopefully not only help the company make better architectural decisions in the future, resulting in a more competitive product that will require less developer and architecture work hours to maintain and evolve with new features.

6.6.2 The Tools Developed

In order to extract the data used in this thesis a tool was developed to extract a model of the architecture as well as measure the different quality metrics form the model. The tool developed have been designed to extract this model based on the implemented architecture at the company, and therefore make use of implementation specific attributes and can unfortunately therefore not be shared with this thesis report. Both due to the fact that the source code could reveal sensitive information about the company and its architecture, and the tools developed would not be usable on any other architectural implementation developed somewhere else. The tools and the source code for them was handed over to the company upon the conclusion of this thesis work and will hopefully help the company reduce the level of erosion present in their implemented architecture, as well as help them in their migration work.

6.6.3 Ethics and Confidentiality

Many steps have been taken to make sure that the data used in this study, both the data gathered from the company's implemented architecture and the data gathered through developers interviews were collected and used in a ethical way. The selection of interview subjects was made without revealing their identity to any others at the company, all interviews data was recorded and transcribed with the names redacted. The interview subjects were also contacted after the transcription process to review the transcript and conclusions taken form the interview, in order to ensure that the number of misunderstandings are as low as possible. The identity of the company, as well as any detail of the company's application and architecture not necessary to understand the results and conclusions of this thesis. Without these considerations this thesis paper risks harming either the company's application or the company's employees integrity.

6.7 Threats to Validity, Reliability and Replicability

This section seeks to critically examine the validity, reliability and replicability of the methods used and results collected in this thesis.

6.7.1 The Static Architectural Analysis

While the metrics that the tools developed during this thesis to measure the architectural quality during the static analysis have been explained in the method chapter and related to the literature that served as a basis for the metrics and methods used. The fact that the tools developed had to be created to specifically model and measure the quality of the implemented architecture belonging to the company that the thesis was performed at, this threatens the replicability as this prevents the tools developed to be used in any other context. Nor can the code be included as an attachment to provide greater insight into how the architectural quality metrics were gathered as the source code would threaten the integrity of the company's architecture and product. Some of the metrics used were also company specific, such as SCA and EV however architecture specific compliance metrics are not unusual when performing static architectural analysis and them metrics are explained in enough detail to apply to other architectures.

6.7.2 Developer Interviews

As shown in table 4.2 the interview subjects were selected to give a diverse set of views on the architecture. The primarily factor selected for was the amount of time the interview subject have been working at the company. The primary group of developer roles that the interviews subjects were selected from was backend developers, as the architecture under study was a backend architecture. Some fullstack developers, developers that work on both the frontend application and the backend architecture, were also included as well as one frontend developers to ensure that some frontend prescriptive on the architecture were not overlooked. It is important to select interview subjects based on their differences to ensure the validity of the data collected during a qualitative study [40]. While the distribution of subject work experience is very good, the validity of this study would improve if more interview subjects with none-backend roles were interviewed. It was also decided that the software architects at the company should be excluded from the interviews since their perspective was accounted for through the static architectural analysis. However, their perspective would provide a good point of comparison during the interview conclusion gathering process.
In Runeson and Hösts paper on the subject of conducting case studies[40] it is mentioned that the number of interview subjects should lead to data saturation. Data saturation in this context means that additional interviews no longer will yield any new information. While 12 interview subjects are not a low number for a project of this size, it is still believed by this thesis writer that full data saturation was not archived. Some particular topics where never fully explored, for example, the causes for the few dips in the overall levels of coupling found in the architectures history. It was never fully determined if this was an intentional redesign or just a consequence of some components being removed. Therefore, a higher number of interview subjects with the appropriate domain knowledge to answer these questions would be helpful to improve validity of this thesis project.

A semi-structure interview structure was chosen where not all questions were asked in the same way and in the same order, some of the questions had to be customized depending on the development team the developer belonged to and could not be documented in the appendix A.2 in order to maintain the confidentiality of the company's code. These factors harm the replicability of the study since in order to recreate this study, even at the same company, a researcher would have to have access to more information then provided in this report.

6.7.3 Microservice Reengineering

As mentioned in section 6.4, the reengineering methodology is based on methodology found in literature but since these methods are primarily based on experts making design decision. Since the methods were applied on the architecture by the thesis writer it introduces two primary problems. It threatens the *Reliability* of the results as the thesis writer's architectural knowledge is limited as opposed if an expert were about to migrate the same architecture. It also threatens the *Replicability* as the same methods performed by differently skilled people and on different architectures could result in a significantly different result.

6.8 Future Work

As mentioned in the introduction chapter, the company motivated its migration towards a microservice architecture in the hope that it would be more modular and erosion tolerant then their current service-oriented implementation. While this thesis has assessed the level of erosion and some of the causes of this erosion in the current implementation and left some possible suggestions for microservice migrations and some of their effects the actual effects when implemented are outside of the scope of this thesis. However, a useful follow up study

would be documenting the actual effects of the migration on architectural quality, what was improved, what challenges were encountered and how where they resolved? There is currently not much research on the topic on actual microservice implementation and comparing the actual effects of the migration to the theoretical one presented in this report.
Another possible future topic would be tracking the erosion and evolution of the future implemented microservice to compare that result to the results of this thesis, did the microservice implementation improve maintainability and modularity? In fact, this was the original intended topic of this thesis, as the company is developing another application parallel to the one studied in this thesis using a microservice architecture. The original topic of the thesis was to study and compare the evolution and erosion between them but as the microservice based application was still in an early in development and thus unsuited for study.
There is also a lack of research erosion and evolution's on the specific topic of service-oriented architecture, so more studies on the effects of erosion on service oriented and microservice architectures as well as other maintenance related issues.

7 Conclusion

This chapter will conclude this thesis by summarizing the conclusions drawn as they relate to the research questions posed in section 1.3.

*"**How can architectural evolution and erosion of architectural qualities be measured when analyzing a continuously developed service-oriented architecture?**"*
As concluded by assessing the needs, design goals and the available compiled source code that were available for study, a combination of methods was selected. A combination of a static architectural analysis method for measuring architectural quality metrics and an interview based exploratory methodology to find the reasons for the change in measured quality metrics where deemed most appropriate. The static architectural tool developed measures compliance to architectural design rules, detects structural problems, and architectural coupling through dependencies. The developer interviews discovered some of the underlaying reasons for the changing quality metrics by investigating the developers background, architectural knowledge, development and maintenance process. As well as perspective on specific architectural development events or architectural anomalies. This conclusion is based on the data available at the company, the design qualities the company saw as most important and the methods found in relevant literature.

*"**What architectural problems and challenges appears when continuously developing a service-oriented architecture based on OSGi?**"*
During the architecture under study's lifetime, several challenges have appeared during its development. As outlined in section 5.3, all measured complexity metrics have significantly increased over the one year of development history this study has tracked. The number of consistency violations have more than doubled with an increase 140%. The number of measured structural problems have gone from none to 45, and the level of coupling have increased by 41%. These increases in the measured metrics having occurred alongside only a 17% increase in the number of components that the architecture consists of.
When causes of these changes in quality metrics where investigated through developer interviews, as outlined and future expanded upon in sections 5.5 and 6.1, the large number of conclusions related to the architectures quality and development. These conclusions can be read in full in section 5.5 but in conclusion, the primary challenges discovered consisted of the following. The challenge of spreading up to date architectural knowledge when the

architectures requirements and design is changed. It was discovered that some parts of the original architectural designs had been abandoned during is evolution. This was primary cause of the increase of the number of *Consistency violations* and *exposure violations*. As the architectural documentation and design was not kept up to date with this change there existed some discrepancy between how different teams decided to follow or disregard certain architectural design decision as well as how they familiarize themselves with the architecture. There also exists some unclarity as to how to deal with new architectural challenges that the disregard of the original design introduces.To reveres this, it is possible to update architectural design and updating the architectural documentation and implementing architectural compliance monitoring into the test suites or architectural quality evaluation tools. Another related challenge to this is the monitoring, usage and knowledge of a certain security interface feature. It was found that the primary reasons for the increasing amount of *Special Component Access Violations* is the lack of knowledge of its existence and usage, as well as a mistrust of some developers towards the interface component believing it to do more harm than good. These violations are more concerning as they threaten the stability of the application running within the architecture. This problem can to be mitigated with future updates to documentation and architectural education, and developing monitoring tools to detect when the interface component is not used and finally addressing the problems the developers have with the component. Another challenge is the balancing between the coupling of components and code reuse, as the coupling was introducing further difficulties when updating and removing components from the implemented architecture and that is often solved through work around solution that primarily consist of adding additional duplicate components. This can be problematic as it adds additional complexity to the architecture. These challenges can be mitigated by future prioritizing architectural maintenance tasks and introducing dependency monitoring into the developers and architect's suite of tools. The balance of the maintenance of architectural quality and the development of new functionality and other development condition related issues were also discovered during the interviews. Such issues include the lack of a proper test suite and testing data for some components that prevents the developers to properly test components before release, the release cycle that can rush the development of some components, and the low priority given to the maintenance and refactoring of issues not directly related to the fulfilment of new functional requirements.

"***What effects can a migration towards a microservice architecture have on this issues and overall architectural quality?***"
As shown in section 5.6 it was determined that even using a simple migration clustering based migration method for the loosely coupled and easily migrated components. The number of consistency violations where decreased by 51%, structural violations and the level of coupling by 47%. Future migration of the existing architecture can archive even more benefits in quality metrics. It was also determined through a study of existing software architecture literature, future outlined in sections 6.2 and 6.3, that this architectural migration together with some erosion prevention and repair methods could mitigate future architectural erosion and reverse the erosion currently present in the implemented architecture. Some of the steps have been covered in the conclusions of the previous research question like the taking steps to reduce the architectural knowledge gap, including complete regression testing of components as a migration criterion, increasing the priority of early maintenance of known issues and creating architectural compliance and dependency monitoring tools for developers and architects. But also, some microservice specific design decisions, future expanded upon in section 6.5, like breaking away from shared resources, like databases, and migrating that functionality into the microservice components to make components less coupled and more modular. The act of setting up dependencies between microservice components should be treated more seriously than before and should be treated as an architectural design decision, alternate code sharing methods should be considered such as the usage of binary dependencies and code cloning in order to maintain the microservice components independence. In

conclusion, a migration to microservice architectural style can be used to improve desired architectural quality metrics by primary moving complexities from the architectural level into the components themselves, providing an opportunity for architectural redevelopment, establishing migration criteria on all components that take into account some of problems that appeared during the evolution of the current architecture like the sometimes inadequate suite of automated tests and the usage of appropriate interface components when interfacing with support systems, and finally allowing for components that are independent of each other to run in independent instances.

Bibliography

[1] Khubaib Amjad Alam, Rodina Ahmad, Adnan Akhunzada, Mohd Hairul Nizam Md Nasir, and Samee U Khan. "Impact analysis and change propagation in service-oriented enterprises: A systematic review". In: *Information Systems* 54 (2015), pp. 43–73.

[2] OSGi Alliance. *About the OSGi Service Platform, Technical Whitepaper Revision 4.1, 7 June 2007*. 2007.

[3] OSGi Alliance. *Bundle headers reference - OSGi Alliance*. read 2017-02-22 08:33. 2017. URL: https://www.osgi.org/bundle-headers-reference/.

[4] OSGi Alliance. *OSGi Alliance about us page*. read 2017-02-21 14:01. 2017. URL: https://www.osgi.org/about-us/.

[5] OSGi Alliance. "OSGi Service Platform–Service Compendium (Release 4, Version 4.2)". In: *Createspace, August* (2009).

[6] Armin Balalaie, Abbas Heydarnoori, and Pooyan Jamshidi. "Microservices architecture enables DevOps: migration to a cloud-native architecture". In: *IEEE Software 33.3* (2016).

[7] Olivier Barais, Anne Françoise Le Meur, Laurence Duchien, and Julia Lawall. "Software architecture evolution". In: *Software Evolution*. Springer, 2008, pp. 233–262.

[8] Graph Drawing Steering Committee. *The GraphML File Format*. read 2017-05-06 15:54. 2017. URL: http://graphml.graphdrawing.org/.

[9] Lakshitha De Silva and Dharini Balasubramaniam. "Controlling software architecture erosion: A survey". In: *Journal of Systems and Software* 85.1 (2012), pp. 132–151.

[10] Liliana Dobrica and Eila Niemelä. "A survey on software architecture analysis methods". In: *IEEE Transactions on software Engineering* 28.7 (2002), pp. 638–653.

[11] Stephen G Eick, Todd L Graves, Alan F Karr, J Steve Marron, and Audris Mockus. "Does code decay? assessing the evidence from change management data". In: *IEEE Transactions on Software Engineering* 27.1 (2001), pp. 1–12.

[12] Daniel Escobar, Diana Cárdenas, Rolando Amarillo, Eddie Castro, Kelly Garcés, Carlos Parra, and Rubby Casallas. "Towards the understanding and evolution of monolithic applications as microservices". In: *Computing Conference (CLEI), 2016 XLII Latin American*. IEEE. 2016, pp. 1–11.

[13] Roy T Fielding and Richard N Taylor. *Architectural styles and the design of network-based software architectures*. University of California, Irvine Doctoral dissertation, 2000.

[14] Roy Fielding, Jim Gettys, Jeffrey Mogul, Henrik Frystyk, Larry Masinter, Paul Leach, and Tim Berners-Lee. *Hypertext transfer protocol–HTTP/1.1*. Tech. rep. 1999.

[15] Ian Gorton. *Essential software architecture*. Springer Science & Business Media, 2006.

[16] Sara Hassan and Rami Bahsoon. “Microservices and Their Design Trade-Offs: A Self-Adaptive Roadmap”. In: *Services Computing (SCC), 2016 IEEE International Conference on*. IEEE. 2016, pp. 813–818.

[17] Reiko Heckel, Rui Correia, Carlos Matos, Mohammad El-Ramly, Georgios Koutsoukos, and Luis Andrade. “Architectural transformations: From legacy to three-tier and services”. In: *Software Evolution*. Springer, 2008, pp. 139–170.

[18] Rich Hilliard. “Ieee-std-1471-2000 recommended practice for architectural description of software-intensive systems”. In: *IEEE, http://standards.ieee.org* 12.16-20 (2000), p. 2000.

[19] Lorin Hochstein and Mikael Lindvall. “Combating architectural degeneration: a survey”. In: *Information and Software Technology* 47.10 (2005), pp. 643–656.

[20] Software Engineering Institute. *Community Software Architecture Definitions*. read 2017-01-24 18:10. 2016. URL: `http://www.sei.cmu.edu/architecture/start/glossary/community.cfm`.

[21] Software Engineering Institute. *Software Engineering Institute homepage*. read 2017-01-24 18:16. 2016. URL: `http://www.sei.cmu.edu/index.cfm`.

[22] Clemente Izurieta and James M Bieman. “A multiple case study of design pattern decay, grime, and rot in evolving software systems”. In: *Software Quality Journal* (2013), pp. 1–35.

[23] Ivar Jacobson and Fredrik Lindström. “Reengineering of old systems to an object-oriented architecture”. In: *ACM Sigplan Notices*. Vol. 26. 11. ACM. 1991, pp. 340–350.

[24] Catherine Blake Jaktman, John Leaney, and Ming Liu. “Structural analysis of the software architecture—a maintenance assessment case study”. In: *Software Architecture*. Springer, 1999, pp. 455–470.

[25] T Karhikeyan and J Geetha. “A metrics suite and fuzzy model for measuring coupling in Service Oriented Architecture”. In: *Recent Advances in Computing and Software Systems (RACSS), 2012 International Conference on*. IEEE. 2012, pp. 254–259.

[26] Jens Knodel, Mikael Lindvall, Dirk Muthig, and Matthias Naab. “Static evaluation of software architectures”. In: *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*. IEEE. 2006, 10–pp.

[27] Rainer Koschke and Daniel Simon. “Hierarchical Reflexion Models.” In: *WCRE*. Vol. 3. 2003, p. 36.

[28] Isela Macia, Roberta Arcoverde, Alessandro Garcia, Christina Chavez, and Arndt von Staa. “On the relevance of code anomalies for identifying architecture degradation symptoms”. In: *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*. IEEE. 2012, pp. 277–286.

[29] Nilo Mitra, Yves Lafon, et al. “Soap version 1.2 part 0: Primer”. In: *W3C recommendation* 24 (2003), p. 12.

[30] Gail C Murphy, David Notkin, and Kevin J. Sullivan. “Software reflexion models: Bridging the gap between design and implementation”. In: *IEEE Transactions on Software Engineering* 27.4 (2001), pp. 364–380.

[31] Sam Neuman. *Building Microservices: Designing Fine-Grained Systems*. O’Reilly Media, 2015.

[32] Liam O'Brien, Paulo Merson, and Len Bass. "Quality attributes for service-oriented architectures". In: *Development in SOA Environments. IEEE Computer Society* (2007).

[33] Mike P. Papazoglou. "Service-oriented computing: Concepts, characteristics and directions". In: *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on* (2003).

[34] Mike P Papazoglou and Willem-Jan Heuvel. "Service oriented architectures: approaches, technologies and research issues". In: *The VLDB Journal—The International Journal on Very Large Data Bases* 16.3 (2007), pp. 389–415.

[35] David Lorge Parnas. "Software aging". In: *Proceedings of the 16th international conference on Software engineering*. IEEE Computer Society Press. 1994, pp. 279–287.

[36] Leonardo Passos, Ricardo Terra, Marco Tulio Valente, Renato Diniz, and Nabor das Chagas Mendonca. "Static architecture-conformance checking: An illustrative overview". In: *IEEE software* 27.5 (2010), pp. 82–89.

[37] Cesare Pautasso, Olaf Zimmermann, Mike Amundsen, James Lewis, and Nicolai Josuttis. "Microservices in Practice, Part 1". In: (2017).

[38] Mikhail Perepletchikov, Caspar Ryan, Keith Frampton, and Zahir Tari. "Coupling metrics for predicting maintainability in service-oriented designs". In: *Software Engineering Conference, 2007. ASWEC 2007. 18th Australian*. IEEE. 2007, pp. 329–340.

[39] Dominik Rost, Matthias Naab, Crescencio Lima, and Christina von Flach Garcia Chavez. "Software architecture documentation for developers: a survey". In: *European Conference on Software Architecture*. Springer. 2013, pp. 72–88.

[40] Per Runeson and Martin Höst. "Guidelines for conducting and reporting case study research in software engineering". In: *Empirical software engineering* 14.2 (2009), p. 131.

[41] Dmitry I Savchenko, Gleb I Radchenko, and Ossi Taipale. "Microservices validation: Mjolnirr platform case study". In: *Information and Communication Technology, Electronics and Microelectronics (MIPRO), 2015 38th International Convention on*. IEEE. 2015, pp. 235–240.

[42] Joachim Schelp and Stephan Aier. "SOA and EA-sustainable contributions for increasing corporate agility". In: *System Sciences, 2009. HICSS'09. 42nd Hawaii International Conference on*. IEEE. 2009, pp. 1–8.

[43] Bingu Shim, Siho Choue, Suntae Kim, and Sooyong Park. "A design quality model for service-oriented architecture". In: *Software Engineering Conference, 2008. APSEC'08. 15th Asia-Pacific*. IEEE. 2008, pp. 403–410.

[44] Alan Sill. "The Design and Architecture of Microservices". In: *IEEE Cloud Computing 3.5 (2016): 76-80* (2016).

[45] Michael Stal. "Using architectural patterns and blueprints for service-oriented architecture". In: *IEEE software* 23.2 (2006), pp. 54–61.

[46] Andre LC Tavares and Marco Tulio Valente. "A gentle introduction to OSGi". In: *ACM SIGSOFT Software Engineering Notes* 33.5 (2008), p. 8.

[47] Mohammad Hadi Valipour, Bavar AmirZafari, Khashayar Niki Maleki, and Negin Daneshpour. "A brief survey of software architecture concepts and service oriented architecture". In: *Computer Science and Information Technology, 2009. ICCSIT 2009. 2nd IEEE International Conference on*. IEEE. 2009, pp. 34–38.

[48] Jilles Van Gurp and Jan Bosch. "Design erosion: problems and causes". In: *Journal of systems and software* 61.2 (2002), pp. 105–119.

[49] Jilles Van Gurp, Jan Bosch, and Sjaak Brinkkemper. "Design erosion in evolving software products". In: *ELISA workshop*. 2003, p. 134.

[50] Jilles Van Gurp, Sjaak Brinkkemper, and Jan Bosch. "Design preservation over subsequent releases of a software product: a case study of Baan ERP". In: *Journal of Software: Evolution and Process* 17.4 (2005), pp. 277–306.

[51] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. "Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud". In: *Computing Colombian Conference (10CCC), 2015 10th*. IEEE. 2015, pp. 583–590.

[52] Shuying Wang and Miriam AM Capretz. "Dependency and entropy based impact analysis for service-oriented system evolution". In: *Proceedings of the 2011 IEEE/WIC/ACM International Conferences on Web Intelligence and Intelligent Agent Technology-Volume 01*. IEEE Computer Society. 2011, pp. 412–417.

[53] Xiao-jun Wang. "Metrics for evaluating coupling and service granularity in service oriented architecture". In: *Information Engineering and Computer Science, 2009. ICIECS 2009. International Conference on*. IEEE. 2009, pp. 1–4.

[54] Rainer Weinreich and Georg Buchgeher. "Towards supporting the software architecture life cycle". In: *Journal of Systems and Software* 85.3 (2012), pp. 546–561.

[55] yWorks. *yEd Graph Editor*. read 2017-05-06 15:54. 2017. URL: https://www.yworks.com/products/yed.

A Appendices

A.1 Interview Protocol

1. **Q1:** Introduction to study
   - Explain how the results of the interview will be used, ask for consent to use the interview in the study.
   - The study is investigating the erosion and evolution of the architecture.
   - These interviews will be used to understand the developers perspective.
2. **Q2:** Background of the interview subject
   - Developer team, position, how long have you worked here?
   - How familiar are you with the architecture?
     - Have you made use of the documentation?
     - Are you familiar with the architectural rules?
3. **Q3:** Architectural considerations during development
   - Tell me about the process of developing new components.
     - have you made use of Templates, Generated code or frameworks?
   - How is the layer placement of new components decided?
     - Have you placed components in multiple layers at the same time?
     - If so, What where the justification for this placement?
   - How do you decide on how many components are needed for some functionality, when do you separate concerns?
   - How do you decide on upon the usage of dependencies between components?
     - Are the dependencies allowed in accordance with the architecture?
   - What consideration goes into using internal or outside support systems?
     - Is the usage of special interface component considered?

- What considerations are made when handling user sessions?
- What considerations are made when exposing functionality to outside systems?

4. **Q4:** Architectural and component maintenance tasks

   - Tell me about the process of maintaining components
     - Is maintenance only done to fulfill new or updated functional and non-functional requirements?
   - Are ever architectural quality taken into account when maintaining components.
     - Are any bug fixes of a architectural nature?
   - Tell me about your experience with the release cycle?
     - Does the release cycle effect component quality?
     - Does the release cycle effect architectural quality?

5. **Q5:** Specific component questions

   - Questions in this phase where specific to the components that the team the interviewee are a part of. The questions can therefor not be recorded in this protocol in order to preserve the integrity of the software discussed.

6. **Q6:** Interview wrap up

   - Thanking the interview subject for participating.
   - Summarize the conclusion of the interview for the subject.
   - Ask the interview subject if he or she would be willing to participate in any follow up interviews if necessary.

A.2 Resulting Quality Metrics from Static Analysis

Version	Bundles	CA	CD	EV	SCA	**SA**	DD	ID	DOC	SD	IDD	IID	**CoA**
0	197	22	0	0	0	0	621	680	82	12	416	672	2483
1	197	22	0	0	0	0	621	680	82	12	416	672	2483
2	197	22	0	0	0	0	621	680	82	12	416	672	2483
3	197	22	0	0	0	0	621	680	82	12	416	672	2483
4	197	22	0	0	0	0	621	680	82	12	416	672	2483
5	195	22	0	0	0	0	618	653	80	12	415	645	2423
6	195	22	0	0	0	0	618	653	80	12	415	645	2423
7	198	23	0	0	0	0	621	669	81	12	419	661	2463
8	198	23	0	0	0	0	621	669	81	12	419	661	2463
9	198	22	0	0	0	0	621	669	81	12	419	661	2463
10	198	22	0	0	0	0	621	669	81	12	419	661	2463
11	198	22	0	0	0	0	622	669	81	12	420	661	2465
12	198	22	0	0	0	0	623	670	81	12	421	662	2469
13	186	36	1	0	7	8	587	679	133	11	414	671	2495
14	186	36	1	0	7	8	587	679	133	11	414	671	2495
15	186	36	1	0	7	8	587	679	133	12	414	671	2496
16	186	36	1	0	7	8	587	679	133	12	414	671	2496
17	186	36	1	0	7	8	587	679	133	12	414	671	2496
18	186	36	1	0	7	8	587	679	134	12	414	671	2497
19	190	37	1	1	8	10	592	679	135	12	420	689	2545
20	190	37	1	1	8	10	592	679	135	12	420	689	2545
21	190	37	1	1	8	10	592	679	135	12	420	689	2545
22	190	37	1	1	8	10	592	679	135	12	420	689	2545
23	190	37	1	1	8	10	592	679	135	12	420	689	2545
24	190	38	0	1	8	9	591	682	135	12	418	674	2512
25	190	38	0	1	8	9	591	682	135	12	418	674	2512
26	190	38	0	1	8	9	591	682	135	12	418	674	2512
27	196	41	1	2	13	16	610	711	139	12	440	701	2613
28	196	41	1	2	13	16	610	711	139	12	440	701	2613
29	196	41	1	2	13	16	610	711	139	12	440	701	2613
30	196	39	1	2	13	16	610	711	139	12	440	701	2613
31	196	41	1	2	13	16	610	711	139	12	440	701	2613
32	206	45	1	6	15	22	628	746	142	12	457	736	2721
33	206	46	1	6	15	22	630	748	143	12	459	737	2729
34	206	46	1	6	15	22	632	760	141	12	461	749	2755
35	206	46	1	6	15	22	632	760	141	12	461	749	2755
36	206	46	1	6	15	22	632	760	141	12	461	749	2755
37	206	46	1	6	15	22	632	760	141	12	461	749	2755
38	206	46	1	6	15	22	632	760	141	12	461	749	2755
39	206	46	1	6	15	22	632	760	141	12	461	749	2755
40	206	46	1	6	15	22	632	760	141	12	461	749	2755
41	206	46	1	6	15	22	632	760	141	12	461	749	2755
42	213	46	1	8	15	24	638	799	144	12	472	788	2853
43	213	46	1	8	15	24	638	799	144	12	472	788	2853
44	213	46	1	8	15	24	638	799	144	12	472	788	2853
45	213	46	1	8	15	24	638	799	144	12	472	788	2853
46	213	46	1	8	15	24	638	799	144	12	472	788	2853
47	213	46	1	8	15	24	638	799	144	12	472	788	2853
48	221	47	1	11	17	29	661	954	149	12	569	943	3288
49	222	47	1	11	17	29	663	954	149	12	571	943	3292
50	222	47	1	11	17	29	663	954	149	12	571	943	3292
51	222	47	1	11	17	29	663	954	149	12	571	943	3292
52	222	47	1	11	17	29	663	954	149	12	571	943	3292
53	222	47	2	11	17	30	664	967	149	12	573	956	3321
54	228	51	2	17	23	42	684	986	151	14	595	975	3405
55	228	51	1	17	23	41	673	964	151	14	583	953	3338
56	231	53	2	18	25	45	691	1025	153	15	602	1014	3500

Table A2T1: Architectural quality metrics gathered from all available versions of the implemented architecture

Migration	OSGi Bundles	MS Component	CA	CD	EV	SCA	**SA**	DD	MD	WD	ID	DOC	SD	IDD	IID	**CoA**	OSGi-U
1	140	20	26	2	10	12	24	332	0	173	368	104	8	442	364	1847	72
2	76	35	20	0	8	7	15	121	42	238	76	77	3	354	76	1075	76
3	0	98	0	0	0	0	0	0	345	0	0	0	0	345	0	858	0

Table A2T2: Architectural quality metrics gathered after microservice migration